



TUTORIAL

Sachin Sirohi

R Introduction

R Programming Tutorial is designed for both beginners and professionals. Our tutorial provides all the basic and advanced concepts of data analysis and visualization.

R is a software environment which is used to analyze statistical information and graphical representation. R allows us to do modular programming using functions.

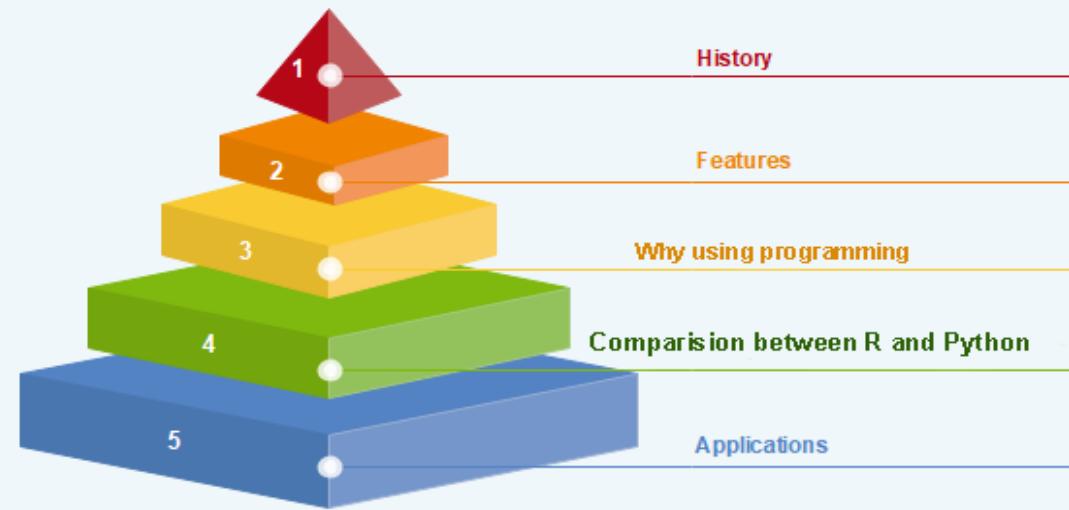
Our R tutorial includes all topics of R such as introduction, features, installation, rstudio ide, variables, datatypes, operators, if statement, vector, data handing, graphics, statistical modelling, etc. This programming language was named R, based on the first name letter of the two authors (Robert Gentleman and Ross Ihaka).

What is R Programming

"**R is an interpreted computer programming language which was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand.**" The **R Development Core Team** currently develops R. It is also a software environment used to analyze **statistical information, graphical representation, reporting, and data modeling**. R is the implementation of the **S programming** language, which is combined with **lexical scoping semantics**.

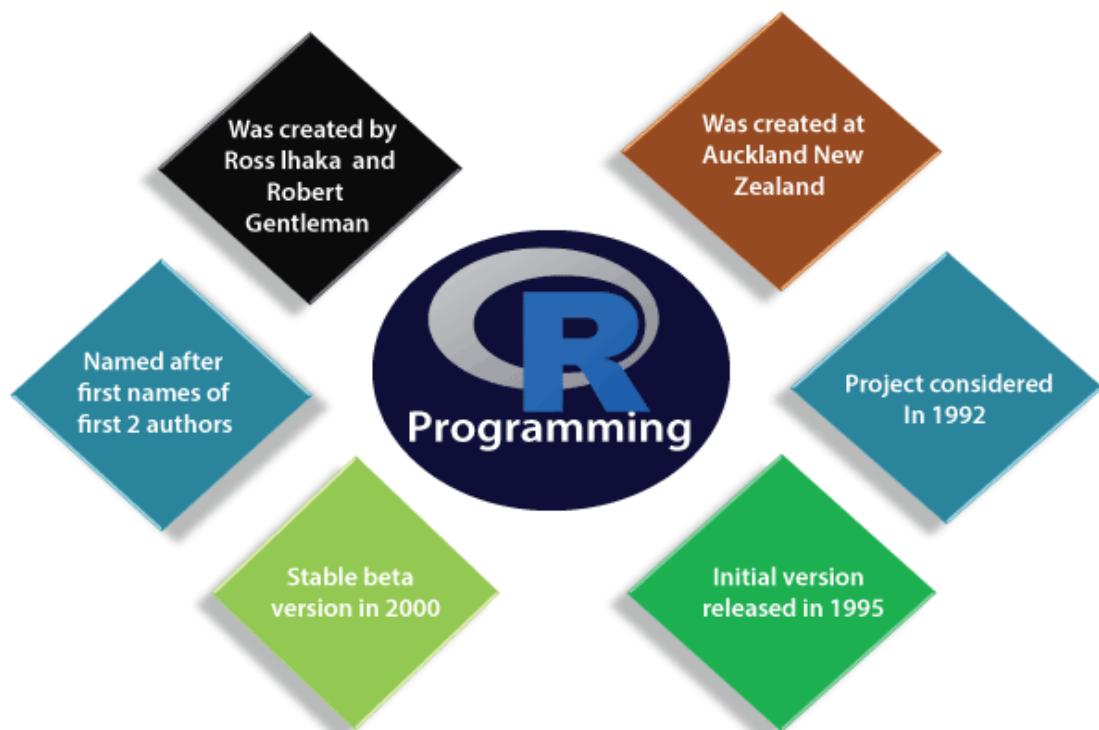
In the present era, R is one of the most important tool which is used by researchers, data analyst, statisticians, and marketers for retrieving, cleaning, analyzing, visualizing, and presenting data.

R Programming



History of R Programming

The history of R goes back about 20-30 years ago. R was developed by Ross Ihaka and Robert Gentleman in the University of Auckland, New Zealand, and the R Development Core Team currently develops it. This programming language name is taken from the name of both the developers. The first project was considered in 1992. The initial version was released in 1995, and in 2000, a stable beta version was released.



The following table shows the release date, version, and description of R language:

Version-Release	Date	Description
0.49	1997-04-23	First time R's source was released, and CRAN (Comprehensive R Archive Network) was started.
0.60	1997-12-05	R officially gets the GNU license.
0.65.1	1999-10-07	update.packages and install.packages both are included.
1.0	2000-02-29	The first production-ready version was released.
1.4	2001-12-19	First version for Mac OS is made available.
2.0	2004-10-04	The first version for Mac OS is made available.
2.1	2005-04-18	Add support for UTF-8encoding, internationalization, localization etc.
2.11	2010-04-22	Add support for Windows 64-bit systems.
2.13	2011-04-14	Added a function that rapidly converts code to byte code.
2.14	2011-10-31	Added some new packages.
2.15	2012-03-30	Improved serialization speed for long vectors.
3.0	2013-04-03	Support for larger numeric values on 64-bit systems.
3.4	2017-04-21	The just-in-time compilation (JIT) is enabled by default.
3.5	2018-04-23	Added new features such as compact internal representation of integer sequences, serialization format etc.

Features of R programming

R is a domain-specific programming language which aims to do data analysis. It has some unique features which make it very powerful. The most important arguably being the notation of vectors. These vectors allow us to perform a complex operation on a set of values in a single command. There are the following features of R programming:

1. It is a simple and effective programming language which has been well developed.
2. It is data analysis software.
3. It is a well-designed, easy, and effective language which has the concepts of user-defined, looping, conditional, and various I/O facilities.
4. It has a consistent and incorporated set of tools which are used for data analysis.
5. For different types of calculation on arrays, lists and vectors, R contains a suite of operators.
6. It provides effective data handling and storage facility.
7. It is an open-source, powerful, and highly extensible software.
8. It provides highly extensible graphical techniques.
9. It allows us to perform multiple calculations using vectors.
10. R is an interpreted language.

Why use R Programming?

There are several tools available in the market to perform data analysis. Learning new languages is time taken. The data scientist can use two excellent tools, i.e., R and Python. We may not have time to learn them both at the time when we get started to learn data science. Learning statistical modeling and algorithm is more important than to learn a programming language. A programming language is used to compute and communicate our discovery.

The important task in data science is the way we deal with the data: clean, feature engineering, feature selection, and import. It should be our primary focus. Data scientist job is to understand the data, manipulate it, and expose the best approach. For machine learning, the best algorithms can be implemented with R. **Keras** and **TensorFlow** allow us to create high-end machine learning techniques. R has a package to perform **Xgboost**. Xgboost is one of the best algorithms for **Kaggle competition**.

R communicate with the other languages and possibly calls Python, Java, C++. The big data world is also accessible to R. We can connect R with different databases like **Spark** or **Hadoop**.

In brief, R is a great tool to investigate and explore the data. The elaborate analysis such as clustering, correlation, and data reduction are done with R.

Comparison between R and Python

Data science deals with identifying, extracting, and representing meaningful information from the data source. R, Python, SAS, SQL, Tableau, MATLAB, etc. are the most useful tools for data science. R and Python are the most used ones. But still, it becomes confusing to choose the better or the most suitable one among the two, R and Python.

Comparison Index	R	Python
Overview	"R is an interpreted computer programming language which was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand ." The R Development Core Team currently develops R. R is also a software environment which is used to analyze statistical information, graphical representation, reporting, and data modeling.	Python is an Interpreted high-level programming language used for general-purpose programming. Guido Van Rossum created it, and it was first released in 1991. Python has a very simple and clean code syntax. It emphasizes the code readability and debugging is also simple and easier in Python.
Specialties for data science	R packages have advanced techniques which are very useful for statistical work. The CRAN text view is provided by many useful R packages. These packages cover everything from Psychometrics to Genetics to Finance.	For finding outliers in a data set both R and Python are equally good. But for developing a web service to allow peoples to upload datasets and find outliers, Python is better.
Functionalities	For data analysis, R has inbuilt functionalities	Most of the data analysis functionalities are not inbuilt. They are available through packages like Numpy and Pandas
Key domains of application	Data visualization is a key aspect of analysis. R packages such as ggplot2, ggvis, lattice, etc. make data visualization easier.	Python is better for deep learning because Python packages such as Caffe, Keras, OpenNN, etc. allows the development of the deep neural network in a very simple way.

Availability of packages	There are hundreds of packages and ways to accomplish needful data science tasks.	Python has few main packages such as viz, Scikit learn, and Pandas for data analysis of machine learning, respectively.
---------------------------------	---	---

Applications of R

There are several applications available in real-time. Some of the popular applications are as follows:

- Facebook
- Google
- Twitter
- HRDAG
- Sunlight Foundation
- RealClimate
- NDAA
- XBOX ONE
- ANZ
- FDA

R Advantages and Disadvantages

R is the most popular programming language for statistical modeling and analysis. Like other programming languages, R also has some advantages and disadvantages. It is a continuously evolving language which means that many cons will slowly fade away with future updates to R.

There are the following pros and cons of R

Pros and Cons of



Advantages



Open Source

Data Wrangling

Array of Packages

Quality Plotting and Graphing

Platform Independent

Machine Learning Operations

Countinuously Growing

Disadvantages

Weak Origin

Data Handling

Basic Security

Complicated Language

Lesser Speed

Pros

1) Open Source

An open-source language is a language on which we can work without any need for a license or a fee. R is an open-source language. We can contribute to the development of R by optimizing our packages, developing new ones, and resolving issues.

2) Platform Independent

R is a platform-independent language or cross-platform programming language which means its code can run on all operating systems. R enables programmers to develop software for several competing platforms by writing a program only once. R can run quite easily on Windows, Linux, and Mac.

3) Machine Learning Operations

R allows us to do various machine learning operations such as

ification and regression. For this purpose, R provides various packages and features for developing the artificial neural network. R is used by the best data scientists in the world.

4) Exemplary support for data wrangling

R allows us to perform data wrangling. R provides packages such as dplyr, readr which are capable of transforming messy data into a structured form.

5) Quality plotting and graphing

R simplifies quality plotting and graphing. R libraries such as ggplot2 and plotly advocates for visually appealing and aesthetic graphs which set R apart from other programming languages.

6) The array of packages

R has a rich set of packages. R has over 10,000 packages in the CRAN repository which are constantly growing. R provides packages for data science and machine learning operations.

7) Statistics

R is mainly known as the language of statistics. It is the main reason why R is predominant than other programming languages for the development of statistical tools.

8) Continuously Growing

R is a constantly evolving programming language. Constantly evolving means when something evolves, it changes or develops over time, like our taste in music and clothes, which evolve as we get older. R is a state of the art which provides updates whenever any new feature is added.

Cons

1) Data Handling

In R, objects are stored in physical memory. It is in contrast with other programming languages like Python. R utilizes more memory as compared to Python. It requires the entire data in one single place which is in the memory. It is not an ideal option when we deal with Big Data.

2) Basic Security

R lacks basic security. It is an essential part of most programming languages such as Python. Because of this, there are many restrictions with R as it cannot be embedded in a web-application.

3) Complicated Language

R is a very complicated language, and it has a steep learning curve. The people who don't have prior knowledge or programming experience may find it difficult to learn R.

4) Weak Origin

The main disadvantage of R is, it does not have support for dynamic or 3D graphics. The reason behind this is its origin. It shares its origin with a much older programming language "S."

5) Lesser Speed

R programming language is much slower than other programming languages such as MATLAB and Python. In comparison to other programming language, R packages are much slower.

In R, algorithms are spread across different packages. The programmers who have no prior knowledge of packages may find it difficult to implement algorithms.

R Packages

R packages are the collection of R functions, sample data, and compile codes. In the R environment, these packages are stored under a directory called "**library**." During installation, R installs a set of packages. We can add packages later when they are needed for some specific purpose. Only the default packages will be available when we start the R console. Other packages which are already installed will be loaded explicitly to be used by the R program.

There is the following list of commands to be used to check, verify, and use the R packages.



1. Check available R package

2. Getting list of all installed packages

3. Install a new Package

4. Load package to library

Install directly
from CRAN

Install package
manually

Check Available R Packages

To check the available R Packages, we have to find the library location in which R packages are contained. R provides `libPaths()` function to find the library locations.

1. `libPaths()`

When the above code executes, it produces the following project, which may vary depending on the local settings of our PCs & Laptops.

```
[1] "C:/Users/ajeet/OneDrive/Documents/R/win-library/3.6"  
[2] "C:/Program Files/R/R-3.6.1/library"
```

Getting the list of all the packages installed

R provides `library()` function, which allows us to get the list of all the installed packages.

1. `library()`

When we execute the above function, it produces the following result, which may vary depending on the local settings of our PCs or laptops.

Packages in library 'C:/Program Files/R/R-3.6.1/library':

```
R packages available

File Edit

Packages in library 'C:/Program Files/R/R-3.6.1/library':


base          The R Base Package
boot          Bootstrap Functions (Originally by Angelo Canty
              for S)
class         Functions for Classification
cluster       "Finding Groups in Data": Cluster Analysis
              Extended Rousseeuw et al.
codetools     Code Analysis Tools for R
compiler      The R Compiler Package
datasets      The R Datasets Package
foreign       Read Data Stored by 'Minitab', 'S', 'SAS',
              'SPSS', 'Stata', 'Systat', 'Weka', 'dBase', ...
graphics      The R Graphics Package
grDevices    The R Graphics Devices and Support for Colours
              and Fonts
grid          The Grid Graphics Package
KernSmooth   Functions for Kernel Smoothing Supporting Wand
              & Jones (1995)
lattice       Trellis Graphics for R
MASS          Support Functions and Datasets for Venables and
              Ripley's MASS
Matrix        Sparse and Dense Matrix Classes and Methods
methods       Formal Methods and Classes
mgcv          Mixed GAM Computation Vehicle with Automatic
```

Like `library()` function, R provides `search()` function to get all packages currently loaded in the R environment.

1. `search()`

When we execute the above code, it will produce the following result, which may vary depending on the local settings of our PCs and laptops:

```
[1] ".GlobalEnv"           "package:stats"       "package:graphics"
[4] "package:grDevices"    "package:utils"        "package:datasets"
[7] "package:methods"      "Autoloads"          "package:base"
```

Install a New Package

In R, there are two techniques to add new R packages. The first technique is installing package directly from the CRAN directory, and the second one is to install it manually after downloading the package to our local system.

Install directly from CRAN

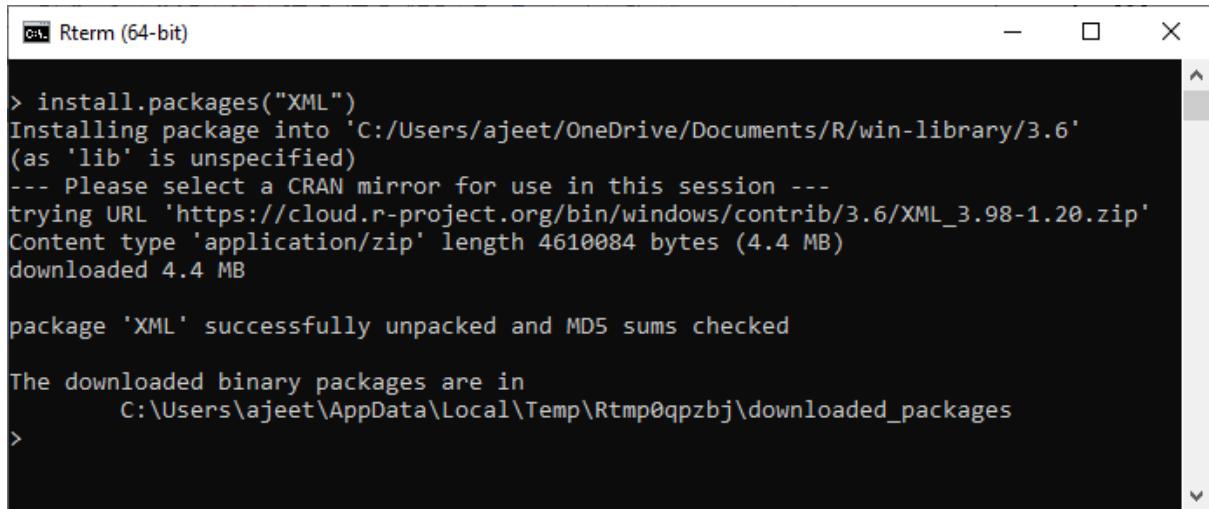
The following command is used to get the packages directly from CRAN webpage and install the package in the R environment. We may be prompted to choose the nearest mirror. Choose the one appropriate to our location.

1. `install.packages("Package Name")`

The syntax of installing XML package is as follows:

1. `install.packages("XML")`

Output



A screenshot of the Rterm (64-bit) window. The title bar says "Rterm (64-bit)". The main area contains the following R session output:

```
> install.packages("XML")
Installing package into 'C:/Users/ajeet/OneDrive/Documents/R/win-library/3.6'
(as 'lib' is unspecified)
--- Please select a CRAN mirror for use in this session ---
trying URL 'https://cloud.r-project.org/bin/windows/contrib/3.6/XML_3.98-1.20.zip'
Content type 'application/zip' length 4610084 bytes (4.4 MB)
downloaded 4.4 MB

package 'XML' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
      C:\Users\ajeet\AppData\Local\Temp\Rtmp0qpzbj\downloaded_packages
>
```

Install package manually

To install a package manually, we first have to download it from https://cran.r-project.org/web/packages/available_packages_by_name.html. The required package will be saved as a .zip file in a suitable location in the local system.

A3 Accurate, Adaptable, and Accessible Error Metrics for Predictive Models

aaSEA Amino Acid Substitution Effect Analyser

abbyyR Access to Abbyy Optical Character Recognition (OCR) API

abc Tools for Approximate Bayesian Computation (ABC)

abc.data Data Only: Tools for Approximate Bayesian Computation (ABC)

ABC.RAP Array Based CpG Region Analysis Pipeline

ABCanalysis Computed ABC Analysis

abcdeFBA ABCDE_FBA: A-Biologist-Can-Do-Everything of Flux Balance Analysis with this package

ABCOptim Implementation of Artificial Bee Colony (ABC) Optimization

ABCn2 Approximate Bayesian Computational Model for Estimating

Once the downloading has finished, we will use the following command:

1. `install.packages(file_name_with_path, repos = NULL, type = "source")`

Install the package named "XML"

1. `install.packages("C:\Users\ajeet\OneDrive\Desktop\graphics\xml2_1.2.2.zip", repos = NULL, type = "source")`

Load Package to Library

We cannot use the package in our code until it will not be loaded into the current R environment. We also need to load a package which is already installed previously but not available in the current environment.

There is the following command to load a package:

1. `library("package Name", lib.loc = "path to library")`

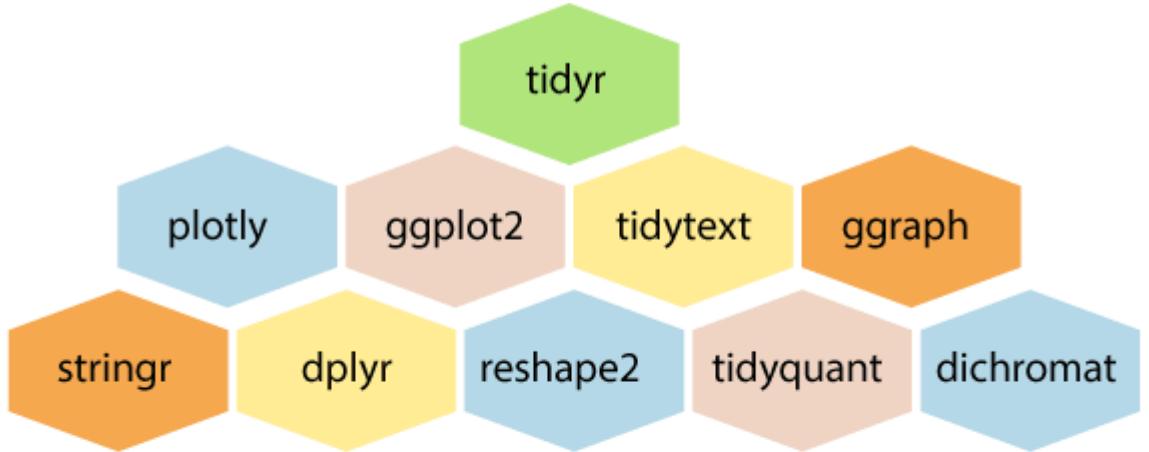
Command to load the XML package

1. `install.packages("C:\Users\ajeet\OneDrive\Desktop\graphics\xml2_1.2.2.zip", repos = NULL, type = "source")`

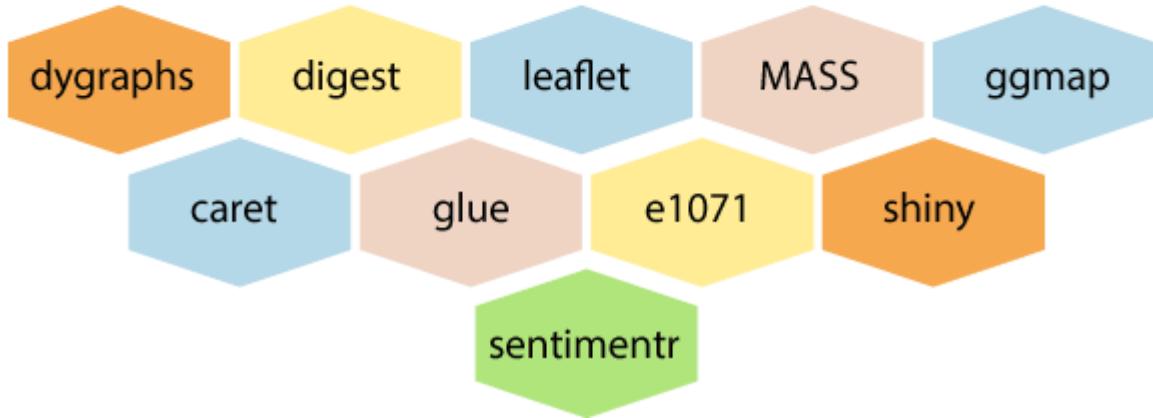
List of R packages

R is the language of data science which includes a vast repository of packages. These packages appeal to different regions which use R for their data purposes. CRAN has 10,000 packages, making it an ocean of superlative statistical work. There are lots of packages in R, but we will discuss the important one.

There are some mostly used and popular packages which are as follows:



list of Packages



1) **tidyverse**

The word **tidyverse** comes from the word **tidy**, which means clear. So the **tidyverse** package is used to make the data 'tidy'. This package works well with **dplyr**. This package is an evolution of the **reshape2** package.

2) **ggplot2**

R allows us to create graphics declaratively. R provides the **ggplot** package for this purpose. This package is famous for its elegant and quality graphs which sets it apart from other visualization packages.

3) **ggraph**

R provides an extension of ggplot known as **ggraph**. The limitation of **ggplot** is the dependency on tabular data is taken away in ggraph.

4) **dplyr**

R allows us to perform data wrangling and data analysis. R provides the **dplyr** library for this purpose. This library facilitates several functions for the data frame in R.

5) **tidyquant**

The tidyquant is a financial package which is used for carrying out quantitative financial analysis. This package adds to the **tidyverse** universe as a financial package which is used for importing, analyzing and visualizing the data.

6) **dygraphs**

The dygraphs package provides an interface to the main JavaScript library which we can use for charting. This package is essentially used for plotting time-series data in R.

7) **leaflet**

For creating interactive visualization, R provides the **leaflet** package. This package is an open-source JavaScript library. The world's popular websites like the New York Times, Github and Flicker, etc. are using leaflet. The leaflet package makes it easier to interact with these sites.

8) **ggmap**

For delineating spatial visualization, the **ggmap** package is used. It is a mapping package which consists of various tools for geolocating and routing.

9) **glue**

R provides the **glue** package to perform the operations of data wrangling. This package is used for evaluating R expressions which are present within the string.

10) shiny

R allows us to develop interactive and aesthetically pleasing web apps by providing a **shiny** package. This package provides various extensions with HTML widgets, CSS, and JavaScript.

11) plotly

The **plotly** package provides online interactive and quality graphs. This package extends upon the JavaScript library **-plotly.js**.

12) tidytext

The **tidytext** package provides various functions of text mining for word processing and carrying out analysis through ggplot, dplyr, and other miscellaneous tools.

13) stringr

The **stringr** package provides simplicity and consistency to use wrappers for the '**stringi**' package. The **stringi** package facilitates common string operations.

14) reshape2

This package facilitates flexible reorganization and aggregation of data using melt () and decast () functions.

15) dichromat

The R **dichromat** package is used to remove Red-Green or Blue-Green contrasts from the colors.

16) digest

The **digest** package is used for the creation of cryptographic hash objects of R functions.

17) MASS

The **MASS** package provides a large number of statistical functions. It provides datasets that are in conjunction with the book "Modern Applied Statistics with S."

18) caret

R allows us to perform classification and regression tasks by providing the caret package. **CaretEnsemble** is a feature of caret which is used for the combination of different models.

19) e1071

The **e1071** library provides useful functions which are essential for data analysis like Naive Bayes, Fourier Transforms, SVMs, Clustering, and other miscellaneous functions.

20) sentimentr

The sentiment package provides functions for carrying out sentiment analysis. It is used to calculate text polarity at the sentence level and to perform aggregation by rows or grouping variables.

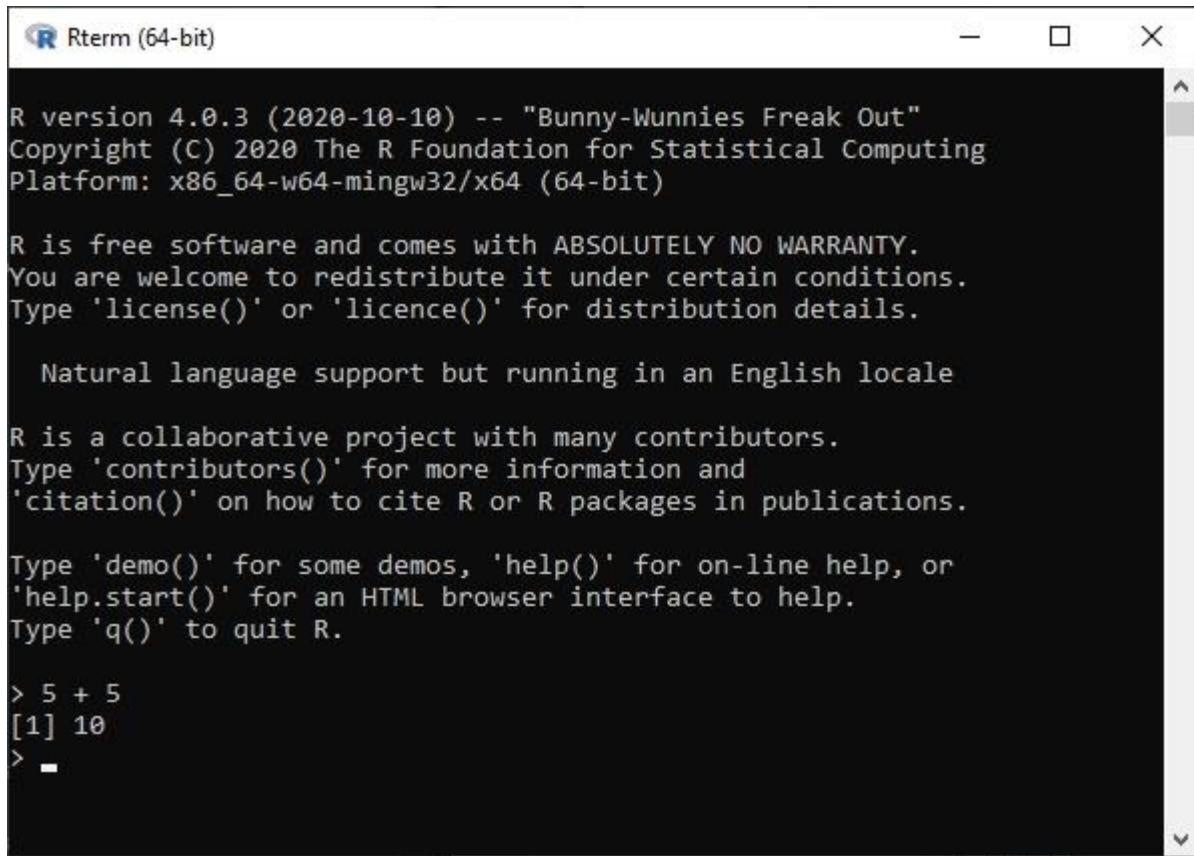
R Get Started

How to Install R

To install R, go to <https://cloud.r-project.org/> and download the latest version of R for Windows, Mac or Linux.

When you have downloaded and installed R, you can run R on your computer.

The screenshot below shows how it may look like when you run R on a Windows PC:



R version 4.0.3 (2020-10-10) -- "Bunny-Wunnies Freak Out"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

```
> 5 + 5
[1] 10
> -
```

If you type **5 + 5**, and press enter, you will see that R outputs **10**.

Learning R

When learning R at W3Schools.com, you can use our "Try it Yourself" tool, which shows both the code and the result in your browser. This will make it easier for you to test and understand every part as we move forward:

Example

5 + 5

Result:

[1] 10

R Syntax

Syntax

To output text in R, use single or double quotes:

Example

```
"Hello World!"
```

To output numbers, just type the number (without quotes):

Example

```
5  
10  
25
```

To do simple calculations, add numbers together:

Example

```
5 + 5
```

R Print Output

Print

Unlike many other programming languages, you can output code in R without using a print function:

Example

```
"Hello World!"
```

However, R does have a `print()` function available if you want to use it. This might be useful if you are familiar with other programming languages, such as [Python](#), which often uses the `print()` function to output code.

Example

```
print("Hello World!")
```

And there are times you must use the `print()` function to output code, for example when working with for loops (which you will learn more about in a later chapter):

Example

```
for (x in 1:10) {  
  print(x)  
}
```

Conclusion: It is up to you whether you want to use the `print()` function to output code. However, when your code is inside an R expression (e.g. inside curly braces `{}` like in the example above), use the `print()` function to output the result.

R Comments

Comments

Comments can be used to explain R code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Comments starts with a `#`. When executing code, R will ignore anything that starts with `#`.

This example uses a comment before a line of code:

Example

```
# This is a comment  
"Hello World!"
```

This example uses a comment at the end of a line of code:

Example

```
"Hello World!" # This is a comment
```

Comments does not have to be text to explain the code, it can also be used to prevent R from executing the code:

Example

```
# "Good morning!"  
"Good night!"
```

Multiline Comments

Unlike other programming languages, such as [Java](#), there are no syntax in R for multiline comments. However, we can just insert a `#` for each line to create multiline comments:

Example

```
# This is a comment  
# written in  
# more than just one line  
"Hello World!"
```

R Variables

Creating Variables in R

Variables are containers for storing data values.

R does not have a command for declaring a variable. A variable is created the moment you first assign a value to it. To assign a value to a variable, use the `<-` sign. To output (or print) the variable value, just type the variable name:

Example

```
name <- "John"  
age <- 40  
  
name # output "John"  
age # output 40
```

From the example above, `name` and `age` are **variables**, while `"John"` and `40` are **values**.

In other programming language, it is common to use `=` as an assignment operator. In R, we can use both `=` and `<-` as assignment operators.

However, `<-` is preferred in most cases because the `=` operator can be forbidden in some context in R.

Print / Output Variables

Compared to many other programming languages, you do not have to use a function to print/output variables in R. You can just type the name of the variable:

Example

```
name <- "John Doe"  
  
name # auto-print the value of the name variable
```

However, R does have a `print()` function available if you want to use it. This might be useful if you are familiar with other programming languages, such as [Python](#), which often use a `print()` function to output variables.

Example

```
name <- "John Doe"  
  
print(name) # print the value of the name variable
```

And there are times you must use the `print()` function to output code, for example when working with `for` loops (which you will learn more about in a later chapter):

Example

```
for (x in 1:10) {  
  print(x)  
}
```

Conclusion: It is up to your if you want to use the `print()` function or not to output code. However, when your code is inside an R expression (for example inside curly braces `{}` like in the example above), use the `print()` function if you want to output the result.

Concatenate Elements

You can also concatenate, or join, two or more elements, by using the `paste()` function.

To combine both text and a variable, R uses comma (,):

Example

```
text <- "awesome"  
  
paste("R is", text)
```

You can also use , to add a variable to another variable:

Example

```
text1 <- "R is"  
text2 <- "awesome"  
  
paste(text1, text2)
```

For numbers, the + character works as a mathematical operator:

Example

```
num1 <- 5  
num2 <- 10  
  
num1 + num2
```

If you try to combine a string (text) and a number, R will give you an error:

Example

```
num <- 5  
text <- "Some text"  
  
num + text
```

Result:

```
Error in num + text : non-numeric argument to binary operator
```

Multiple Variables

R allows you to assign the same value to multiple variables in one line:

Example

```
# Assign the same value to multiple variables in one line
var1 <- var2 <- var3 <- "Orange"

# Print variable values
var1
var2
var3
```

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for R variables are:

- A variable name must start with a letter and can be a combination of letters, digits, period(.) and underscore(_). If it starts with period(.), it cannot be followed by a digit.
- A variable name cannot start with a number or underscore (_)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- Reserved words cannot be used as variables (TRUE, FALSE, NULL, if...)

```
# Legal variable names:
```

```
myvar <- "John"
my_var <- "John"
myVar <- "John"
MYVAR <- "John"
myvar2 <- "John"
.myvar <- "John"
```

```
# Illegal variable names:
```

```
2myvar <- "John"
my-var <- "John"
my var <- "John"
_my_var <- "John"
my_v@ar <- "John"
TRUE <- "John"
```

Remember that variable names are case-sensitive!

R Data Types

Data Types

In programming, data type is an important concept.

Variables can store data of different types, and different types can do different things.

In R, variables do not need to be declared with any particular type, and can even change type after they have been set:

Example

```
my_var <- 30 # my_var is type of numeric  
my_var <- "Sally" # my_var is now of type character (aka string)
```

R has a variety of data types and object classes. You will learn much more about these as you continue to get to know R.

Basic Data Types

Basic data types in R can be divided into the following types:

- **numeric** - (10.5, 55, 787)
- **integer** - (1L, 55L, 100L, where the letter "L" declares this as an integer)
- **complex** - (9 + 3i, where "i" is the imaginary part)
- **character** (a.k.a. string) - ("k", "R is exciting", "FALSE", "11.5")
- **logical** (a.k.a. boolean) - (TRUE or FALSE)

We can use the `class()` function to check the data type of a variable:

Example

```
# numeric  
x <- 10.5  
class(x)  
  
# integer  
x <- 1000L
```

```
class(x)

# complex
x <- 9i + 3
class(x)

# character/string
x <- "R is exciting"
class(x)

# logical/boolean
x <- TRUE
class(x)
```

R Numbers

Numbers

There are three number types in R:

- `numeric`
- `integer`
- `complex`

Variables of number types are created when you assign a value to them:

Example

```
x <- 10.5    # numeric
y <- 10L      # integer
z <- 1i       # complex
```

Numeric

A `numeric` data type is the most common type in R, and contains any number with or without a decimal, like: 10.5, 55, 787:

Example

```
x <- 10.5
y <- 55

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```

Integer

Integers are numeric data without decimals. This is used when you are certain that you will never create a variable that should contain decimals. To create an `integer` variable, you must use the letter `L` after the integer value:

Example

```
x <- 1000L
y <- 55L

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```

Complex

A `complex` number is written with an "`i`" as the imaginary part:

Example

```
x <- 3+5i
y <- 5i

# Print values of x and y
x
```

```
y

# Print the class name of x and y
class(x)
class(y)
```

Type Conversion

You can convert from one type to another with the following functions:

- `as.numeric()`
- `as.integer()`
- `as.complex()`

Example

```
x <- 1L # integer
y <- 2 # numeric

# convert from integer to numeric:
a <- as.numeric(x)

# convert from numeric to integer:
b <- as.integer(y)

# print values of x and y
x
y

# print the class name of a and b
class(a)
class(b)
```

R Math

Simple Math

In R, you can use **operators** to perform common mathematical operations on numbers.

The `+` operator is used to add together two values:

Example

```
10 + 5
```

And the `-` operator is used for subtraction:

Example

```
10 - 5
```

You will learn more about available operators in our [R Operators Tutorial](#).

Built-in Math Functions

R also has many built-in math functions that allows you to perform mathematical tasks on numbers.

For example, the `min()` and `max()` functions can be used to find the lowest or highest number in a set:

Example

```
max(5, 10, 15)
```

```
min(5, 10, 15)
```

sqrt()

The `sqrt()` function returns the square root of a number:

Example

```
sqrt(16)
```

abs()

The `abs()` function returns the absolute (positive) value of a number:

Example

```
abs(-4.7)
```

ceiling() and floor()

The `ceiling()` function rounds a number upwards to its nearest integer, and the `floor()` function rounds a number downwards to its nearest integer, and returns the result:

Example

```
ceiling(1.4)
```

```
floor(1.4)
```

R Strings

String Literals

Strings are used for storing text.

A string is surrounded by either single quotation marks, or double quotation marks:

"hello" is the same as 'hello':

Example

```
"hello"  
'hello'
```

Assign a String to a Variable

Assigning a string to a variable is done with the variable followed by the `<-` operator and the string:

Example

```
str <- "Hello"  
str # print the value of str
```

Multiline Strings

You can assign a multiline string to a variable like this:

Example

```
str <- "Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua."  
  
str # print the value of str
```

However, note that R will add a "\n" at the end of each line break. This is called an escape character, and the **n** character indicates a **new line**.

If you want the line breaks to be inserted at the same position as in the code, use the `cat()` function:

Example

```
str <- "Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua."  
  
cat(str)
```

String Length

There are many useful string functions in R.

For example, to find the number of characters in a string, use the `nchar()` function:

Example

```
str <- "Hello World!"  
  
nchar(str)
```

Check a String

Use the `grep1()` function to check if a character or a sequence of characters are present in a string:

Example

```
str <- "Hello World!"
```

```
grep1("H", str)
grep1("Hello", str)
grep1("X", str)
```

Combine Two Strings

Use the `paste()` function to merge/concatenate two strings:

Example

```
str1 <- "Hello"
str2 <- "World"
```

```
paste(str1, str2)
```

Escape Characters

To insert characters that are illegal in a string, you must use an escape character.

An escape character is a backslash `\` followed by the character you want to insert.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

Example

```
str <- "We are the so-called "Vikings", from the north."
```

```
str
```

Result:

```
Error: unexpected symbol in "str <- "We are the so-called
"Vikings"
```

To fix this problem, use the escape character `\":`

Example

The escape character allows you to use double quotes when you normally would not be allowed:

```
str <- "We are the so-called \"Vikings\", from the north."
```

```
str  
cat(str)
```

Note that auto-printing the **str** variable will print the backslash in the output. You can use the **cat()** function to print it without backslash.

Other escape characters in R:

Code	Result
\\"	Backslash
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace

R Booleans / Logical Values

Booleans (Logical Values)

In programming, you often need to know if an expression is **true** or **false**.

You can evaluate any expression in R, and get one of two answers, **TRUE** or **FALSE**.

When you compare two values, the expression is evaluated and R returns the logical answer:

Example

```
10 > 9      # TRUE because 10 is greater than 9
10 == 9     # FALSE because 10 is not equal to 9
10 < 9      # FALSE because 10 is greater than 9
```

You can also compare two variables:

Example

```
a <- 10
b <- 9

a > b
```

You can also run a condition in an **if** statement, which you will learn much more about in the [if..else](#) chapter.

Example

```
a <- 200
b <- 33

if (b > a) {
  print ("b is greater than a")
} else {
  print("b is not greater than a")
}
```

R Operators

Operators

Operators are used to perform operations on variables and values.

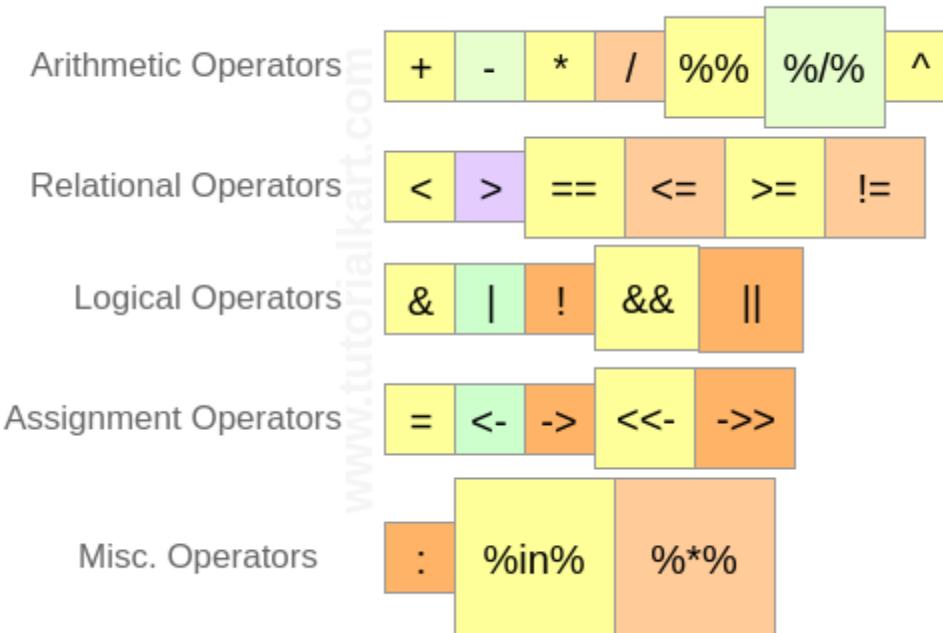
In the example below, we use the `+` operator to add together two values:

Example

`10 + 5`

R divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Miscellaneous operators



R Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	Name	Example
+	Addition	$x + y$
-	Subtraction	$x - y$
*	Multiplication	$x * y$
/	Division	x / y
^	Exponent	$x ^ y$
%%	Modulus (Remainder from division)	$x \% \% y$
%/%	Integer Division	$x \% / \% y$

R Assignment Operators

Assignment operators are used to assign values to variables:

Example

```
my_var <- 3
```

```
my_var <<- 3
```

```
3 -> my_var  
  
3 ->> my_var  
  
my_var # print my_var
```

Note: `<-` is a global assigner. You will learn more about this in the [Global Variable chapter](#).

It is also possible to turn the direction of the assignment operator.

`x <- 3` is equal to `3 -> x`

R Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
<code>==</code>	Equal	<code>x == y</code>
<code>!=</code>	Not equal	<code>x != y</code>
<code>></code>	Greater than	<code>x > y</code>
<code><</code>	Less than	<code>x < y</code>
<code>>=</code>	Greater than or equal to	<code>x >= y</code>
<code><=</code>	Less than or equal to	<code>x <= y</code>

R Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description
&	Element-wise Logical AND operator. It returns TRUE if both elements are TRUE
&&	Logical AND operator - Returns TRUE if both statements are TRUE
	Elementwise- Logical OR operator. It returns TRUE if one of the statement is TRUE
	Logical OR operator. It returns TRUE if one of the statement is TRUE.
!	Logical NOT - returns FALSE if statement is TRUE

R Miscellaneous Operators

Miscellaneous operators are used to manipulate data:

Operator	Description	Example
:	Creates a series of numbers in a sequence	x <- 1:10
%in%	Find out if an element belongs to a vector	x %in% y
%*%	Matrix Multiplication	x <- Matrix1 %*% Matrix2

R If ... Else

Conditions and If Statements

R supports the usual logical conditions from mathematics:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y

<	Less than	$x < y$
\geq	Greater than or equal to	$x \geq y$
\leq	Less than or equal to	$x \leq y$

These conditions can be used in several ways, most commonly in "if statements" and loops.

The if Statement

An "if statement" is written with the `if` keyword, and it is used to specify a block of code to be executed if a condition is `TRUE`:

Example

```
a <- 33
b <- 200

if (b > a) {
  print("b is greater than a")
}
```

In this example we use two variables, `a` and `b`, which are used as a part of the if statement to test whether `b` is greater than `a`. As `a` is `33`, and `b` is `200`, we know that `200` is greater than `33`, and so we print to screen that "b is greater than a".

R uses curly brackets `{ }` to define the scope in the code.

Else If

The `else if` keyword is R's way of saying "if the previous conditions were not true, then try this condition":

Example

```
a <- 33
b <- 33

if (b > a) {
  print("b is greater than a")
} else if (a == b) {
  print ("a and b are equal")
}
```

In this example `a` is equal to `b`, so the first condition is not true, but the `else if` condition is true, so we print to screen that "a and b are equal".

You can use as many `else if` statements as you want in R.

If Else

The `else` keyword catches anything which isn't caught by the preceding conditions:

Example

```
a <- 200
b <- 33

if (b > a) {
  print("b is greater than a")
} else if (a == b) {
  print("a and b are equal")
} else {
  print("a is greater than b")
}
```

In this example, `a` is greater than `b`, so the first condition is not true, also the `else if` condition is not true, so we go to the `else` condition and print to screen that "a is greater than b".

You can also use `else` without `else if`:

Example

```
a <- 200
b <- 33

if (b > a) {
  print("b is greater than a")
} else {
  print("b is not greater than a")
}
```

Nested If Statements

You can also have `if` statements inside `if` statements, this is called *nested if* statements.

Example

```
x <- 41

if (x > 10) {
  print("Above ten")
  if (x > 20) {
    print("and also above 20!")
  } else {
    print("but not above 20.")
  }
} else {
  print("below 10.")
}
```

AND

The `&` symbol (and) is a logical operator, and is used to combine conditional statements:

Example

Test if a is greater than b, AND if c is greater than a:

```
a <- 200
b <- 33
c <- 500
```

```
if (a > b & c > a) {  
  print("Both conditions are true")  
}
```

OR

The `|` symbol (or) is a logical operator, and is used to combine conditional statements:

Example

Test if a is greater than b, or if c is greater than a:

```
a <- 200  
b <- 33  
c <- 500  
  
if (a > b | a > c) {  
  print("At least one of the conditions is true")  
}
```

R While Loop

Loops

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

R has two loop commands:

- `while` loops
- `for` loops

R While Loops

With the `while` loop we can execute a set of statements as long as a condition is TRUE:

Example

Print `i` as long as `i` is less than 6:

```
i <- 1
while (i < 6) {
  print(i)
  i <- i + 1
}
```

In the example above, the loop will continue to produce numbers ranging from 1 to 5. The loop will stop at 6 because `6 < 6` is FALSE.

The `while` loop requires relevant variables to be ready, in this example we need to define an indexing variable, `i`, which we set to 1.

Note: remember to increment `i`, or else the loop will continue forever.

Break

With the `break` statement, we can stop the loop even if the while condition is TRUE:

Example

Exit the loop if `i` is equal to 4.

```
i <- 1
while (i < 6) {
  print(i)
  i <- i + 1
  if (i == 4) {
    break
  }
}
```

The loop will stop at 3 because we have chosen to finish the loop by using the `break` statement when `i` is equal to 4 (`i == 4`).

Next

With the `next` statement, we can skip an iteration without terminating the loop:

Example

Skip the value of 3:

```
i <- 0
while (i < 6) {
  i <- i + 1
  if (i == 3) {
    next
  }
  print(i)
}
```

When the loop passes the value 3, it will skip it and continue to loop.

Yahtzee!

If .. Else Combined with a While Loop

To demonstrate a practical example, let us say we play a game of Yahtzee!

Example

Print "Yahtzee!" If the dice number is 6:

```
dice <- 1
while (dice <= 6) {
  if (dice < 6) {
    print("No Yahtzee")
  } else {
    print("Yahtzee!")
  }
  dice <- dice + 1
}
```

R For Loop

For Loops

A `for` loop is used for iterating over a sequence:

Example

```
for (x in 1:10) {  
  print(x)  
}
```

This is less like the `for` keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the `for` loop we can execute a set of statements, once for each item in a vector, array, list, etc..

You will learn about [lists](#) and [vectors](#), etc in a later chapter.

Example

Print every item in a list:

```
fruits <- list("apple", "banana", "cherry")  
  
for (x in fruits) {  
  print(x)  
}
```

Example

Print the number of dices:

```
dice <- c(1, 2, 3, 4, 5, 6)  
  
for (x in dice) {  
  print(x)  
}
```

The `for` loop does not require an indexing variable to set beforehand, like with `while` loops.

Break

With the `break` statement, we can stop the loop before it has looped through all the items:

Example

Stop the loop at "cherry":

```
fruits <- list("apple", "banana", "cherry")  
  
for (x in fruits) {  
  if (x == "cherry") {  
    break  
  }  
  print(x)  
}
```

The loop will stop at "cherry" because we have chosen to finish the loop by using the `break` statement when `x` is equal to "cherry" (`x == "cherry"`).

Next

With the `next` statement, we can skip an iteration without terminating the loop:

Example

Skip "banana":

```
fruits <- list("apple", "banana", "cherry")  
  
for (x in fruits) {  
  if (x == "banana") {  
    next  
  }  
  print(x)  
}
```

When the loop passes "banana", it will skip it and continue to loop.

Yahtzee!

If .. Else Combined with a For Loop

To demonstrate a practical example, let us say we play a game of Yahtzee!

Example

Print "Yahtzee!" If the dice number is 6:

```
dice <- 1:6

for(x in dice) {
  if (x == 6) {
    print(paste("The dice number is", x, "Yahtzee!"))
  } else {
    print(paste("The dice number is", x, "Not Yahtzee"))
  }
}
```

If the loop reaches the values ranging from 1 to 5, it prints "No Yahtzee" and its number. When it reaches the value 6, it prints "Yahtzee!" and its number.

Nested Loops

You can also have a loop inside of a loop:

Example

Print the adjective of each fruit in a list:

```
adj <- list("red", "big", "tasty")

fruits <- list("apple", "banana", "cherry")
for (x in adj) {
  for (y in fruits) {
    print(paste(x, y))
  }
}
```

R Data Structures

R Vectors

Vectors

A vector is simply a list of items that are of the same type.

To combine the list of items to a vector, use the `c()` function and separate the items by a comma.

In the example below, we create a vector variable called `fruits`, that combine strings:

Example

```
# Vector of strings
fruits <- c("banana", "apple", "orange")

# Print fruits
fruits
```

In this example, we create a vector that combines numerical values:

Example

```
# Vector of numerical values
numbers <- c(1, 2, 3)

# Print numbers
numbers
```

To create a vector with numerical values in a sequence, use the `:` operator:

Example

```
# Vector with numerical values in a sequence
numbers <- 1:10

numbers
```

You can also create numerical values with decimals in a sequence, but note that if the last element does not belong to the sequence, it is not used:

Example

```
# Vector with numerical decimals in a sequence
numbers1 <- 1.5:6.5
```

```
numbers1

# Vector with numerical decimals in a sequence where the last element
# is not used
numbers2 <- 1.5:6.3
numbers2
```

Result:

```
[1] 1.5 2.5 3.5 4.5 5.5 6.5
[1] 1.5 2.5 3.5 4.5 5.5
```

In the example below, we create a vector of logical values:

Example

```
# Vector of logical values
log_values <- c(TRUE, FALSE, TRUE, FALSE)

log_values
```

Vector Length

To find out how many items a vector has, use the `length()` function:

Example

```
fruits <- c("banana", "apple", "orange")

length(fruits)
```

Sort a Vector

To sort items in a vector alphabetically or numerically, use the `sort()` function:

Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")
numbers <- c(13, 3, 5, 7, 20, 2)

sort(fruits) # Sort a string
sort(numbers) # Sort numbers
```

Access Vectors

You can access the vector items by referring to its index number inside brackets `[]`. The first item has index 1, the second item has index 2, and so on:

Example

```
fruits <- c("banana", "apple", "orange")  
  
# Access the first item (banana)  
fruits[1]
```

You can also access multiple elements by referring to different index positions with the `c()` function:

Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")  
  
# Access the first and third item (banana and orange)  
fruits[c(1, 3)]
```

You can also use negative index numbers to access all items except the ones specified:

Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")  
  
# Access all items except for the first item  
fruits[c(-1)]
```

Change an Item

To change the value of a specific item, refer to the index number:

Example

```
fruits <- c("banana", "apple", "orange", "mango", "lemon")  
  
# Change "banana" to "pear"  
fruits[1] <- "pear"
```

```
# Print fruits  
fruits
```

Repeat Vectors

To repeat vectors, use the `rep()` function:

Example

Repeat each value:

```
repeat_each <- rep(c(1,2,3), each = 3)  
  
repeat_each
```

Example

Repeat the sequence of the vector:

```
repeat_times <- rep(c(1,2,3), times = 3)  
  
repeat_times
```

Example

Repeat each value independently:

```
repeat_indepent <- rep(c(1,2,3), times = c(5,2,1))  
  
repeat_indepent
```

Generating Sequenced Vectors

One of the examples on top, showed you how to create a vector with numerical values in a sequence with the `:` operator:

Example

```
numbers <- 1:10  
  
numbers
```

To make bigger or smaller steps in a sequence, use the `seq()` function:

Example

```
numbers <- seq(from = 0, to = 100, by = 20)
```

```
numbers
```

Note: The `seq()` function has three parameters: `from` is where the sequence starts, `to` is where the sequence stops, and `by` is the interval of the sequence.

R Lists

Lists

A list in R can contain many different data types inside it. A list is a collection of data which is ordered and changeable.

To create a list, use the `list()` function:

Example

```
# List of strings
thislist <- list("apple", "banana", "cherry")

# Print the list
thislist
```

Access Lists

You can access the list items by referring to its index number, inside brackets. The first item has index 1, the second item has index 2, and so on:

Example

```
thislist <- list("apple", "banana", "cherry")

thislist[1]
```

Change Item Value

To change the value of a specific item, refer to the index number:

Example

```
thislist <- list("apple", "banana", "cherry")
thislist[1] <- "blackcurrant"

# Print the updated list
thislist
```

List Length

To find out how many items a list has, use the `length()` function:

Example

```
thislist <- list("apple", "banana", "cherry")

length(thislist)
```

Check if Item Exists

To find out if a specified item is present in a list, use the `%in%` operator:

Example

Check if "apple" is present in the list:

```
thislist <- list("apple", "banana", "cherry")

"apple" %in% thislist
```

Add List Items

To add an item to the end of the list, use the `append()` function:

Example

Add "orange" to the list:

```
thislist <- list("apple", "banana", "cherry")  
append(thislist, "orange")
```

To add an item to the right of a specified index, add "*after=index number*" in the `append()` function:

Example

Add "orange" to the list after "banana" (index 2):

```
thislist <- list("apple", "banana", "cherry")  
append(thislist, "orange", after = 2)
```

Remove List Items

You can also remove list items. The following example creates a new, updated list without an "apple" item:

Example

Remove "apple" from the list:

```
thislist <- list("apple", "banana", "cherry")  
  
newlist <- thislist[-1]  
  
# Print the new list  
newlist
```

Range of Indexes

You can specify a range of indexes by specifying where to start and where to end the range, by using the `:` operator:

Example

Return the second, third, fourth and fifth item:

```
thislist <-  
list("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")  
  
(thislist)[2:5]
```

Note: The search will start at index 2 (included) and end at index 5 (included).

Remember that the first item has index 1.

Loop Through a List

You can loop through the list items by using a `for` loop:

Example

Print all items in the list, one by one:

```
thislist <- list("apple", "banana", "cherry")  
  
for (x in thislist) {  
  print(x)  
}
```

Join Two Lists

There are several ways to join, or concatenate, two or more lists in R.

The most common way is to use the `c()` function, which combines two elements together:

Example

```
list1 <- list("a", "b", "c")  
list2 <- list(1,2,3)  
list3 <- c(list1,list2)  
  
list3
```

R Matrices

Matrices

A matrix is a two dimensional data set with columns and rows.

A column is a vertical representation of data, while a row is a horizontal representation of data.

A matrix can be created with the `matrix()` function. Specify the `nrow` and `ncol` parameters to get the amount of rows and columns:

Example

```
# Create a matrix  
thismatrix <- matrix(c(1,2,3,4,5,6), nrow = 3, ncol = 2)  
  
# Print the matrix  
thismatrix
```

Note: Remember the `c()` function is used to concatenate items together.

You can also create a matrix with strings:

Example

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow  
= 2, ncol = 2)  
  
thismatrix
```

Access Matrix Items

You can access the items by using `[]` brackets. The first number "1" in the bracket specifies the row-position, while the second number "2" specifies the column-position:

Example

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow  
= 2, ncol = 2)  
  
thismatrix[1, 2]
```

The whole row can be accessed if you specify a comma **after** the number in the bracket:

Example

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow  
= 2, ncol = 2)
```

```
thismatrix[2,]
```

The whole column can be accessed if you specify a comma **before** the number in the bracket:

Example

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)
```

```
thismatrix[,2]
```

Access More Than One Row

More than one row can be accessed if you use the `c()` function:

Example

```
thismatrix <-  
matrix(c("apple", "banana", "cherry", "orange", "grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)  
  
thismatrix[c(1,2),]
```

Access More Than One Column

More than one column can be accessed if you use the `c()` function:

Example

```
thismatrix <-  
matrix(c("apple", "banana", "cherry", "orange", "grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)  
  
thismatrix[, c(1,2)]
```

Add Rows and Columns

Use the `cbind()` function to add additional columns in a Matrix:

Example

```
thismatrix <-  
matrix(c("apple", "banana", "cherry", "orange", "grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)  
  
newmatrix <- cbind(thismatrix,  
c("strawberry", "blueberry", "raspberry"))  
  
# Print the new matrix  
newmatrix
```

Note: The cells in the new column must be of the same length as the existing matrix.

Use the `rbind()` function to add additional rows in a Matrix:

Example

```
thismatrix <-  
matrix(c("apple", "banana", "cherry", "orange", "grape", "pineapple", "pear", "melon", "fig"), nrow = 3, ncol = 3)  
  
newmatrix <- rbind(thismatrix,  
c("strawberry", "blueberry", "raspberry"))  
  
# Print the new matrix  
newmatrix
```

Note: The cells in the new row must be of the same length as the existing matrix.

Remove Rows and Columns

Use the `c()` function to remove rows and columns in a Matrix:

Example

```
thismatrix <-  
matrix(c("apple", "banana", "cherry", "orange", "mango", "pineapple"),  
nrow = 3, ncol = 2)  
  
#Remove the first row and the first column  
thismatrix <- thismatrix[-c(1), -c(1)]
```

```
thismatrix
```

Check if an Item Exists

To find out if a specified item is present in a matrix, use the `%in%` operator:

Example

Check if "apple" is present in the matrix:

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
"apple" %in% thismatrix
```

Number of Rows and Columns

Use the `dim()` function to find the number of rows and columns in a Matrix:

Example

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
dim(thismatrix)
```

Matrix Length

Use the `length()` function to find the dimension of a Matrix:

Example

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)  
length(thismatrix)
```

Total cells in the matrix is the number of rows multiplied by number of columns.

In the example above: Dimension = $2 \times 2 = 4$.

Loop Through a Matrix

You can loop through a Matrix using a `for` loop. The loop will start at the first row, moving right:

Example

Loop through the matrix items and print them:

```
thismatrix <- matrix(c("apple", "banana", "cherry", "orange"), nrow = 2, ncol = 2)

for (rows in 1:nrow(thismatrix)) {
  for (columns in 1:ncol(thismatrix)) {
    print(thismatrix[rows, columns])
  }
}
```

Combine two Matrices

Again, you can use the `rbind()` or `cbind()` function to combine two or more matrices together:

Example

```
# Combine matrices
Matrix1 <- matrix(c("apple", "banana", "cherry", "grape"), nrow = 2, ncol = 2)
Matrix2 <- matrix(c("orange", "mango", "pineapple", "watermelon"), nrow = 2, ncol = 2)

# Adding it as a rows
Matrix_Combined <- rbind(Matrix1, Matrix2)
Matrix_Combined

# Adding it as a columns
Matrix_Combined <- cbind(Matrix1, Matrix2)
Matrix_Combined
```

R Arrays

Arrays

Compared to matrices, arrays can have more than two dimensions.

We can use the `array()` function to create an array, and the `dim` parameter to specify the dimensions:

Example

```
# An array with one dimension with values ranging from 1 to 24
thisarray <- c(1:24)
thisarray

# An array with more than one dimension
multiarray <- array(thisarray, dim = c(4, 3, 2))
multiarray
```

Example Explained

In the example above we create an array with the values 1 to 24.

How does `dim=c(4,3,2)` work?

The first and second number in the bracket specifies the amount of rows and columns.

The last number in the bracket specifies how many dimensions we want.

Note: Arrays can only have one data type.

Access Array Items

You can access the array elements by referring to the index position. You can use the `[]` brackets to access the desired elements from an array:

Example

```
thisarray <- c(1:24)
multiarray <- array(thisarray, dim = c(4, 3, 2))

multiarray[2, 3, 2]
```

The syntax is as follow: `array[row position, column position, matrix level]`

You can also access the whole row or column from a matrix in an array, by using the `c()` function:

Example

```
thisarray <- c(1:24)

# Access all the items from the first row from matrix one
multiarray <- array(thisarray, dim = c(4, 3, 2))
multiarray[c(1),,1]

# Access all the items from the first column from matrix one
multiarray <- array(thisarray, dim = c(4, 3, 2))
```

A comma (,) before c() means that we want to access the column.

A comma (,) after c() means that we want to access the row.

Check if an Item Exists

To find out if a specified item is present in an array, use the `%in%` operator:

Example

Check if the value "2" is present in the array:

```
thisarray <- c(1:24)
multiarray <- array(thisarray, dim = c(4, 3, 2))

2 %in% multiarray
```

Amount of Rows and Columns

Use the `dim()` function to find the amount of rows and columns in an array:

Example

```
thisarray <- c(1:24)
multiarray <- array(thisarray, dim = c(4, 3, 2))

dim(multiarray)
```

Array Length

Use the `length()` function to find the dimension of an array:

Example

```
thisarray <- c(1:24)
multiarray <- array(thisarray, dim = c(4, 3, 2))

length(multiarray)
```

Loop Through an Array

You can loop through the array items by using a `for` loop:

Example

```
thisarray <- c(1:24)
multiarray <- array(thisarray, dim = c(4, 3, 2))

for(x in multiarray){
  print(x)
}
```

R Data Frames

Data Frames

Data Frames are data displayed in a format as a table.

Data Frames can have different types of data inside it. While the first column can be `character`, the second and third can be `numeric` or `logical`. However, each column should have the same type of data.

Use the `data.frame()` function to create a data frame:

Example

```
# Create a data frame
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
```

```
Pulse = c(100, 150, 120),  
Duration = c(60, 30, 45)  
)  
  
# Print the data frame  
Data_Frame
```

Summarize the Data

Use the `summary()` function to summarize the data from a Data Frame:

Example

```
Data_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
Data_Frame  
  
summary(Data_Frame)
```

You will learn more about the `summary()` function in the statistical part of the R tutorial.

Access Items

We can use single brackets `[]`, double brackets `[[]]` or `$` to access columns from a data frame:

Example

```
Data_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
Data_Frame[1]  
  
Data_Frame[["Training"]]  
  
Data_Frame$Training
```

Add Rows

Use the `rbind()` function to add new rows in a Data Frame:

Example

```
Data_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
# Add a new row  
New_row_DF <- rbind(Data_Frame, c("Strength", 110, 110))  
  
# Print the new row  
New_row_DF
```

Add Columns

Use the `cbind()` function to add new columns in a Data Frame:

Example

```
Data_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
# Add a new column  
New_col_DF <- cbind(Data_Frame, Steps = c(1000, 6000, 2000))  
  
# Print the new column  
New_col_DF
```

Remove Rows and Columns

Use the `c()` function to remove rows and columns in a Data Frame:

Example

```
Data_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
# Remove the first row and column  
Data_Frame_New <- Data_Frame[-c(1), -c(1)]  
  
# Print the new data frame  
Data_Frame_New
```

Amount of Rows and Columns

Use the `dim()` function to find the amount of rows and columns in a Data Frame:

Example

```
Data_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
dim(Data_Frame)
```

You can also use the `ncol()` function to find the number of columns and `nrow()` to find the number of rows:

Example

```
Data_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
ncol(Data_Frame)  
nrow(Data_Frame)
```

Data Frame Length

Use the `length()` function to find the number of columns in a Data Frame (similar to `ncol()`):

Example

```
Data_Frame <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
length(Data_Frame)
```

Combining Data Frames

Use the `rbind()` function to combine two or more data frames in R vertically:

Example

```
Data_Frame1 <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),  
  Pulse = c(100, 150, 120),  
  Duration = c(60, 30, 45)  
)  
  
Data_Frame2 <- data.frame (  
  Training = c("Stamina", "Stamina", "Strength"),  
  Pulse = c(140, 150, 160),  
  Duration = c(30, 30, 20)  
)  
  
New_Data_Frame <- rbind(Data_Frame1, Data_Frame2)  
New_Data_Frame
```

And use the `cbind()` function to combine two or more data frames in R horizontally:

Example

```
Data_Frame3 <- data.frame (  
  Training = c("Strength", "Stamina", "Other"),
```

```

Pulse = c(100, 150, 120),
Duration = c(60, 30, 45)
)

Data_Frame4 <- data.frame (
  Steps = c(3000, 6000, 2000),
  Calories = c(300, 400, 300)
)

New_Data_Frame1 <- cbind(Data_Frame3, Data_Frame4)
New_Data_Frame1

```

R Factors

Factors

Factors are used to categorize data. Examples of factors are:

- Demography: Male/Female
- Music: Rock, Pop, Classic, Jazz
- Training: Strength, Stamina

To create a factor, use the `factor()` function and add a vector as argument:

Example

```

# Create a factor
music_genre <-
factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "Jazz"))

# Print the factor
music_genre

```

Result:

```
[1] Jazz     Rock     Classic Classic Pop      Jazz     Rock     Jazz
Levels: Classic Jazz Pop Rock
```

You can see from the example above that that the factor has four levels (categories): Classic, Jazz, Pop and Rock.

To only print the levels, use the `levels()` function:

Example

```
music_genre <-  
factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "  
Jazz"))  
  
levels(music_genre)
```

Result:

```
[1] "Classic" "Jazz"     "Pop"      "Rock"
```

You can also set the levels, by adding the `levels` argument inside the `factor()` function:

Example

```
music_genre <-  
factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "  
Jazz"), levels = c("Classic", "Jazz", "Pop", "Rock", "Other"))  
  
levels(music_genre)
```

Result:

```
[1] "Classic" "Jazz"     "Pop"      "Rock"     "Other"
```

Factor Length

Use the `length()` function to find out how many items there are in the factor:

Example

```
music_genre <-  
factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "  
Jazz"))  
  
length(music_genre)
```

Result:

```
[1] 8
```

Access Factors

To access the items in a factor, refer to the index number, using `[]` brackets:

Example

Access the third item:

```
music_genre <-  
factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "  
Jazz"))  
  
music_genre[3]
```

Result:

```
[1] Classic  
Levels: Classic Jazz Pop Rock
```

Change Item Value

To change the value of a specific item, refer to the index number:

Example

Change the value of the third item:

```
music_genre <-  
factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "  
Jazz"))  
  
music_genre[3] <- "Pop"  
  
music_genre[3]
```

Result:

```
[1] Pop  
Levels: Classic Jazz Pop Rock
```

Note that you cannot change the value of a specific item if it is not already specified in the factor. The following example will produce an error:

Example

Trying to change the value of the third item ("Classic") to an item that does not exist/not predefined ("Opera"):

```
music_genre <-  
factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "  
Jazz"))  
  
music_genre[3] <- "Opera"  
  
music_genre[3]
```

Result:

```
Warning message:  
In `<-.factor`(`*tmp*`, 3, value = "Opera") :  
  invalid factor level, NA generated
```

However, if you have already specified it inside the `levels` argument, it will work:

Example

Change the value of the third item:

```
music_genre <-  
factor(c("Jazz", "Rock", "Classic", "Classic", "Pop", "Jazz", "Rock", "  
Jazz"), levels = c("Classic", "Jazz", "Pop", "Rock", "Opera"))  
  
music_genre[3] <- "Opera"  
  
music_genre[3]
```

Result:

```
[1] Opera  
Levels: Classic Jazz Pop Rock Opera
```

R Functions

A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

Creating a Function

To create a function, use the `function()` keyword:

Example

```
my_function <- function() { # create a function with the name  
  my_function  
  print("Hello World!")  
}
```

Call a Function

To call a function, use the function name followed by parenthesis, like `my_function()`:

Example

```
my_function <- function() {  
  print("Hello World!")  
}  
  
my_function() # call the function named my_function
```

Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (`fname`). When the function is called, we pass along a first name, which is used inside the function to print the full name:

Example

```
my_function <- function(fname) {  
  paste(fname, "Griffin")  
}  
  
my_function("Peter")  
my_function("Lois")  
my_function("Stewie")
```

Parameters or Arguments?

The terms "parameter" and "argument" can be used for the same thing: information that are passed into a function.

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less:

Example

This function expects 2 arguments, and gets 2 arguments:

```
my_function <- function(fname, lname) {  
  paste(fname, lname)  
}  
  
my_function("Peter", "Griffin")
```

If you try to call the function with 1 or 3 arguments, you will get an error:

Example

This function expects 2 arguments, and gets 1 argument:

```
my_function <- function(fname, lname) {  
  paste(fname, lname)  
}  
  
my_function("Peter")
```

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without an argument, it uses the default value:

Example

```
my_function <- function(country = "Norway") {  
  paste("I am from", country)  
}  
  
my_function("Sweden")  
my_function("India")  
my_function() # will get the default value, which is Norway  
my_function("USA")
```

Return Values

To let a function return a result, use the `return()` function:

Example

```
my_function <- function(x) {  
  return (5 * x)  
}  
  
print(my_function(3))  
print(my_function(5))  
print(my_function(9))
```

The output of the code above will be:

```
[1] 15  
[1] 25  
[1] 45
```

Nested Functions

There are two ways to create a nested function:

- Call a function within another function.
- Write a function within a function.

Example

Call a function within another function:

```
Nested_function <- function(x, y) {  
  a <- x + y  
  return(a)  
}
```

```
Nested_function(Nested_function(2,2), Nested_function(3,3))
```

Example Explained

The function tells x to add y.

The first input Nested_function(2,2) is "x" of the main function.

The second input Nested_function(3,3) is "y" of the main function.

The output is therefore $(2+2) + (3+3) = \mathbf{10}$.

Example

Write a function within a function:

```
Outer_func <- function(x) {  
  Inner_func <- function(y) {  
    a <- x + y  
    return(a)  
  }  
  return (Inner_func)  
}  
output <- Outer_func(3) # To call the Outer_func  
output(5)
```

Example Explained

You cannot directly call the function because the Inner_func has been defined (nested) inside the Outer_func.

We need to call Outer_func first in order to call Inner_func as a second step.

We need to create a new variable called output and give it a value, which is 3 here.

We then print the output with the desired value of "y", which in this case is 5.

The output is therefore **8** ($3 + 5$).

Recursion

R also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly, recursion can be a very efficient and mathematically-elegant approach to programming.

In this example, `tri_recursion()` is a function that we have defined to call itself ("recurse"). We use the `k` variable as the data, which decrements (-1) every time we recurse. The recursion ends when the condition is not greater than 0 (i.e. when it is 0).

To a new developer it can take some time to work out how exactly this works, best way to find out is by testing and modifying it.

Example

```
tri_recursion <- function(k) {  
  if (k > 0) {  
    result <- k + tri_recursion(k - 1)  
    print(result)  
  } else {  
    result = 0  
  }  
}
```

```
    return(result)
}
}
tri_recursion(6)
```

R Global Variables

Global Variables

Variables that are created outside of a function are known as **global** variables.

Global variables can be used by everyone, both inside of functions and outside.

Example

Create a variable outside of a function and use it inside the function:

```
txt <- "awesome"
my_function <- function() {
  paste("R is", txt)
}

my_function()
```

If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

Example

Create a variable inside of a function with the same name as the global variable:

```
txt <- "global variable"
my_function <- function() {
  txt = "fantastic"
  paste("R is", txt)
}

my_function()
```

```
txt # print txt
```

If you try to print `txt`, it will return "**global variable**" because we are printing `txt` outside the function.

The Global Assignment Operator

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.

To create a global variable inside a function, you can use the **global assignment** operator `<<-`

Example

If you use the assignment operator `<-`, the variable belongs to the global scope:

```
my_function <- function() {  
  txt <- "fantastic"  
  paste("R is", txt)  
}  
  
my_function()  
  
print(txt)
```

Also, use the **global** assignment operator if you want to change a global variable inside a function:

Example

To change the value of a global variable inside a function, refer to the variable by using the global assignment operator `<<-`:

```
txt <- "awesome"  
my_function <- function() {  
  txt <<- "fantastic"  
  paste("R is", txt)  
}  
  
my_function()  
  
paste("R is", txt)
```

R Graphics

R Plotting Plot

The `plot()` function is used to draw points (markers) in a diagram.

The function takes parameters for specifying points in the diagram.

Parameter 1 specifies points on the **x-axis**.

Parameter 2 specifies points on the **y-axis**.

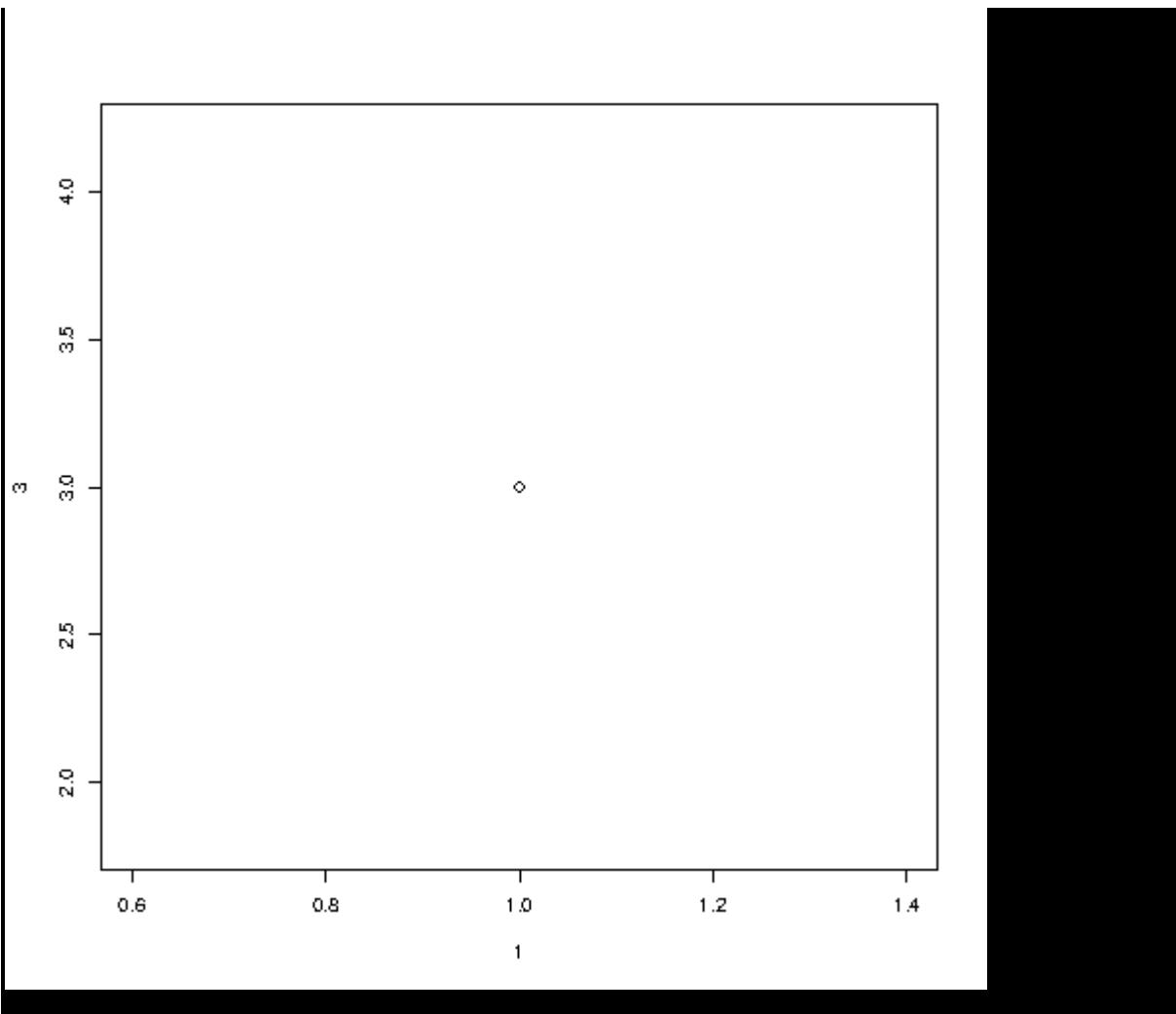
At its simplest, you can use the `plot()` function to plot two numbers against each other:

Example

Draw one point in the diagram, at position (1) and position (3):

```
plot(1, 3)
```

Result:



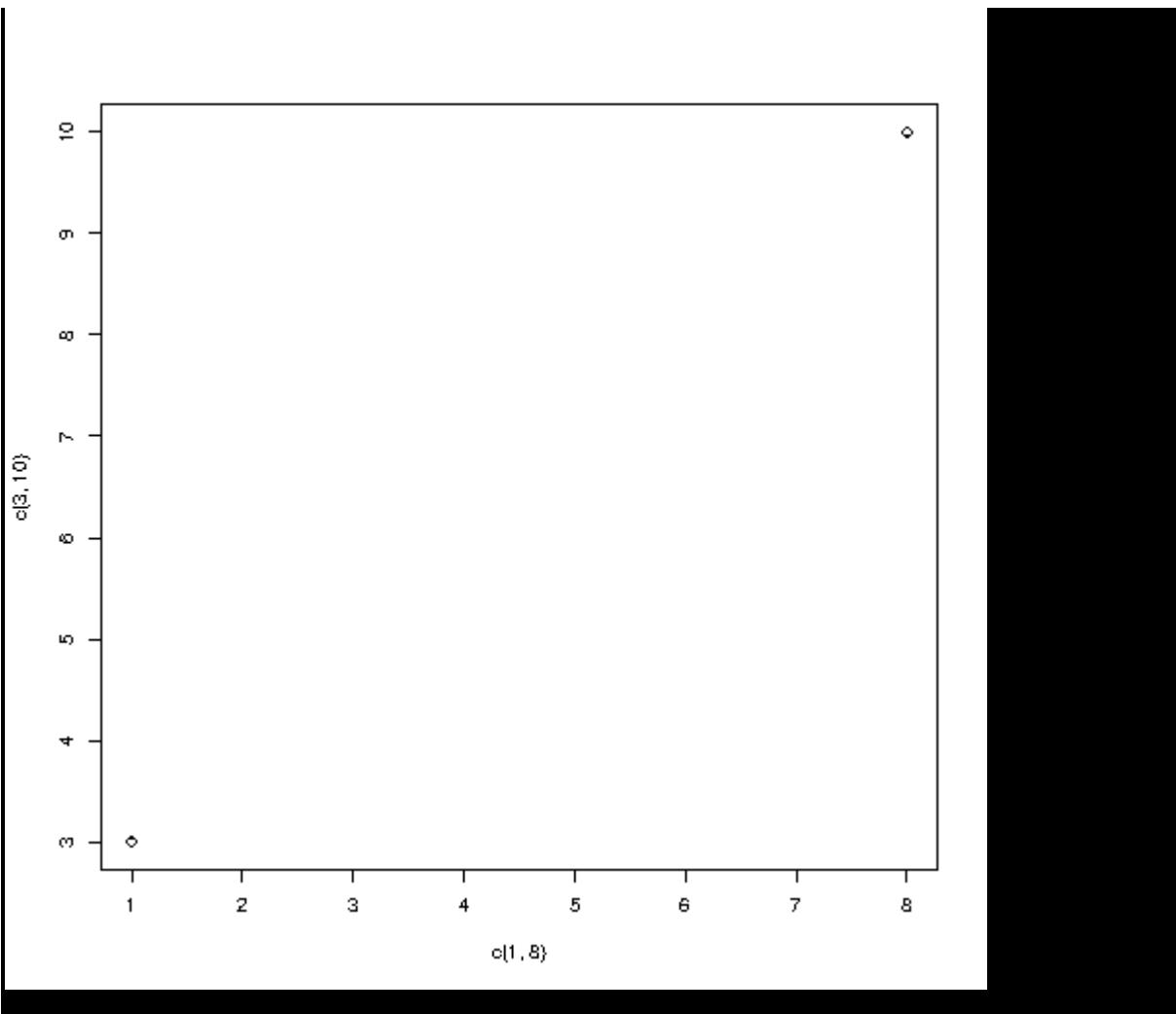
To draw more points, use [vectors](#):

Example

Draw two points in the diagram, one at position (1, 3) and one in position (8, 10):

```
plot(c(1, 8), c(3, 10))
```

Result:



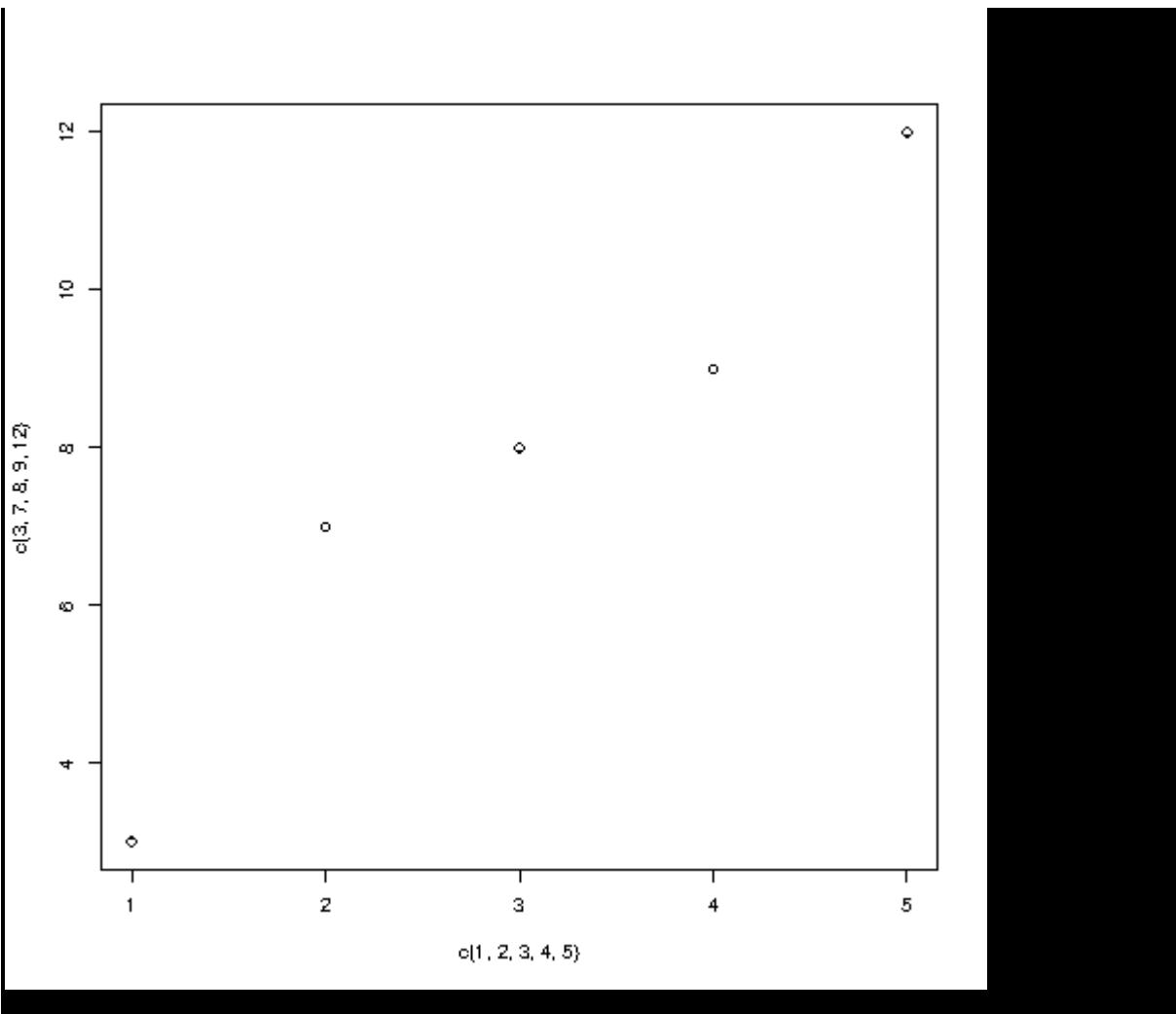
Multiple Points

You can plot as many points as you like, just make sure you have the same number of points in both axis:

Example

```
plot(c(1, 2, 3, 4, 5), c(3, 7, 8, 9, 12))
```

Result:



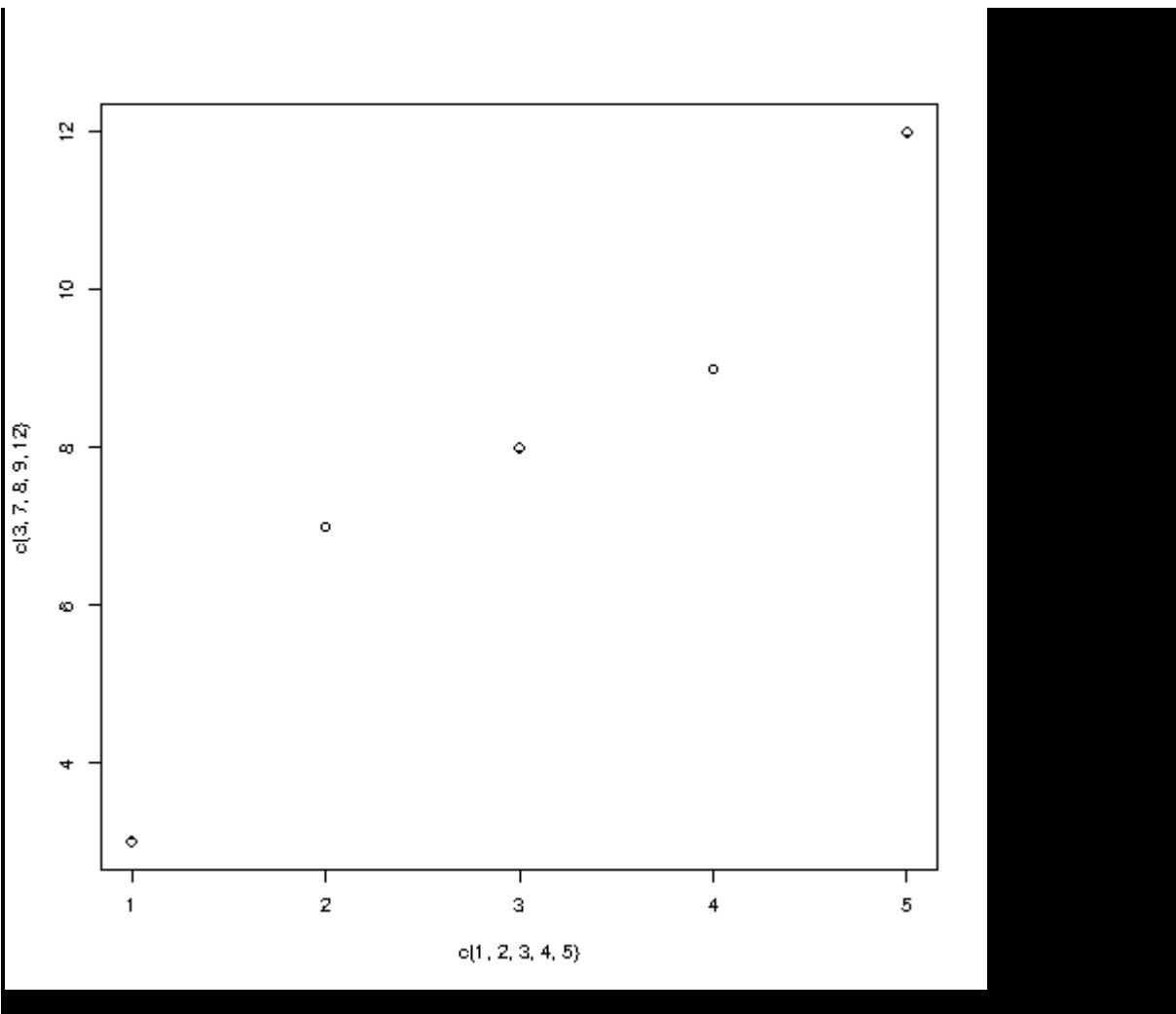
For better organization, when you have many values, it is better to use variables:

Example

```
x <- c(1, 2, 3, 4, 5)
y <- c(3, 7, 8, 9, 12)

plot(x, y)
```

Result:



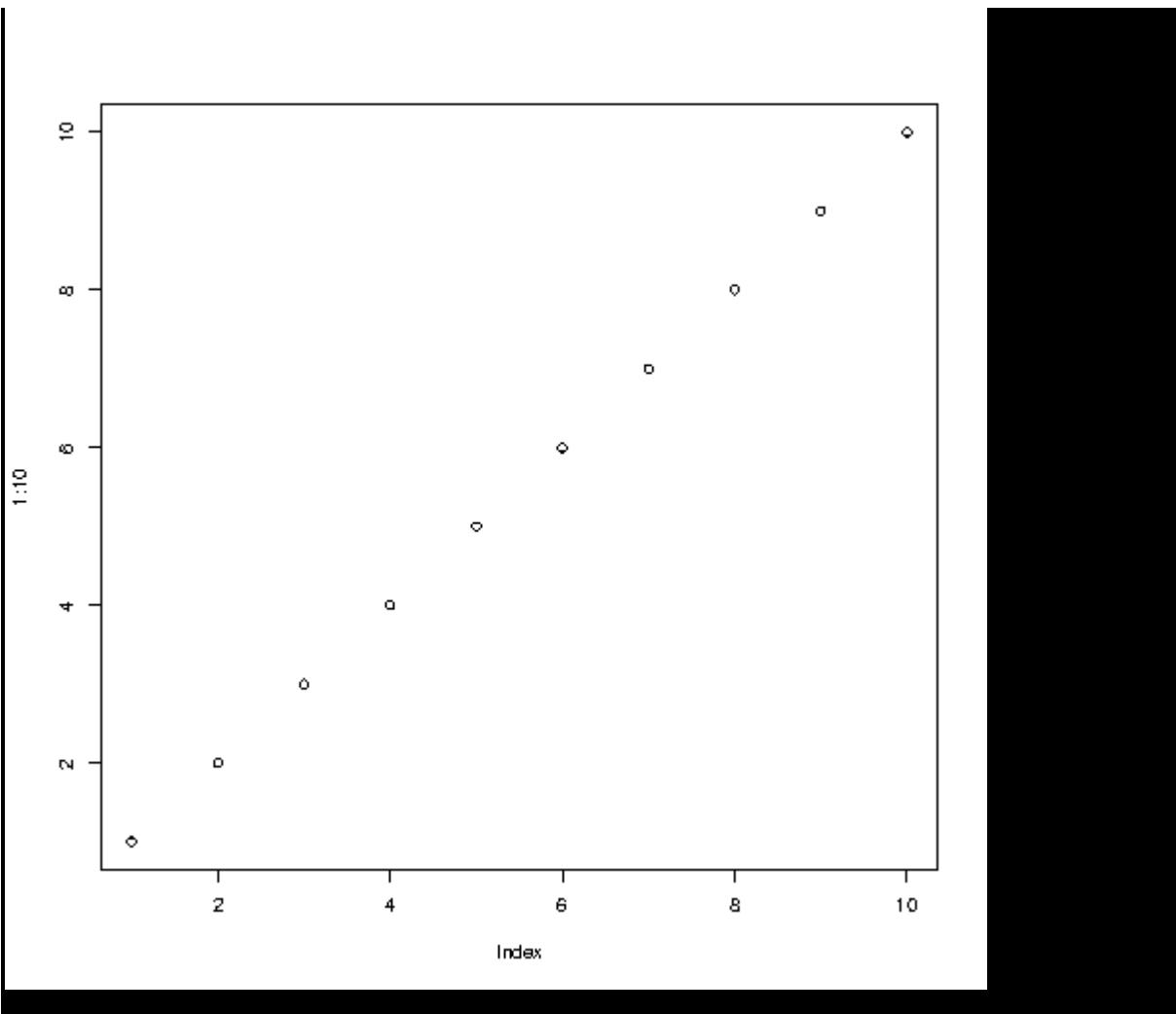
Sequences of Points

If you want to draw dots in a sequence, on both the **x-axis** and the **y-axis**, use the `:` operator:

Example

```
plot(1:10)
```

Result:



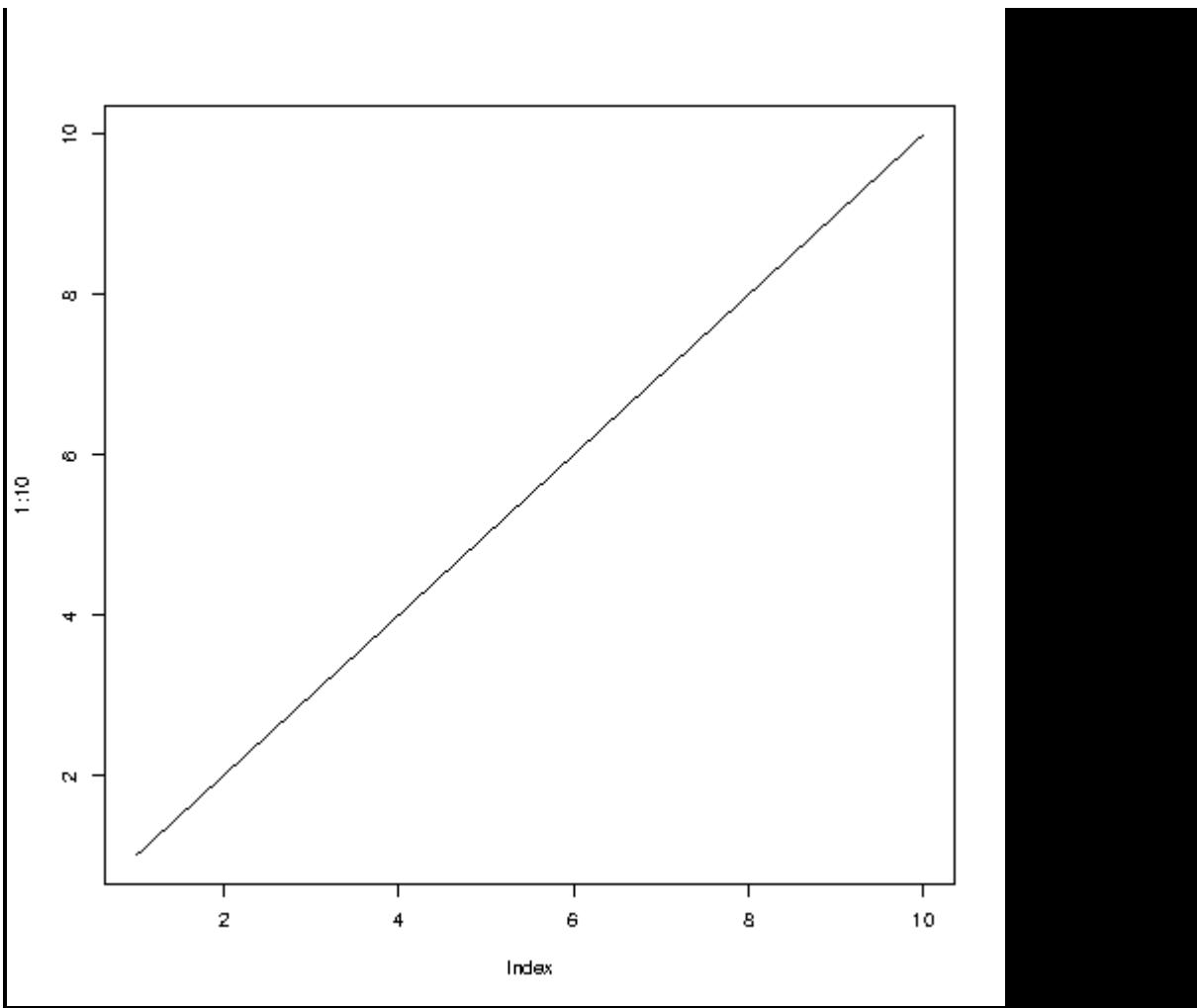
Draw a Line

The `plot()` function also takes a `type` parameter with the value `l` to draw a line to connect all the points in the diagram:

Example

```
plot(1:10, type="l")
```

Result:



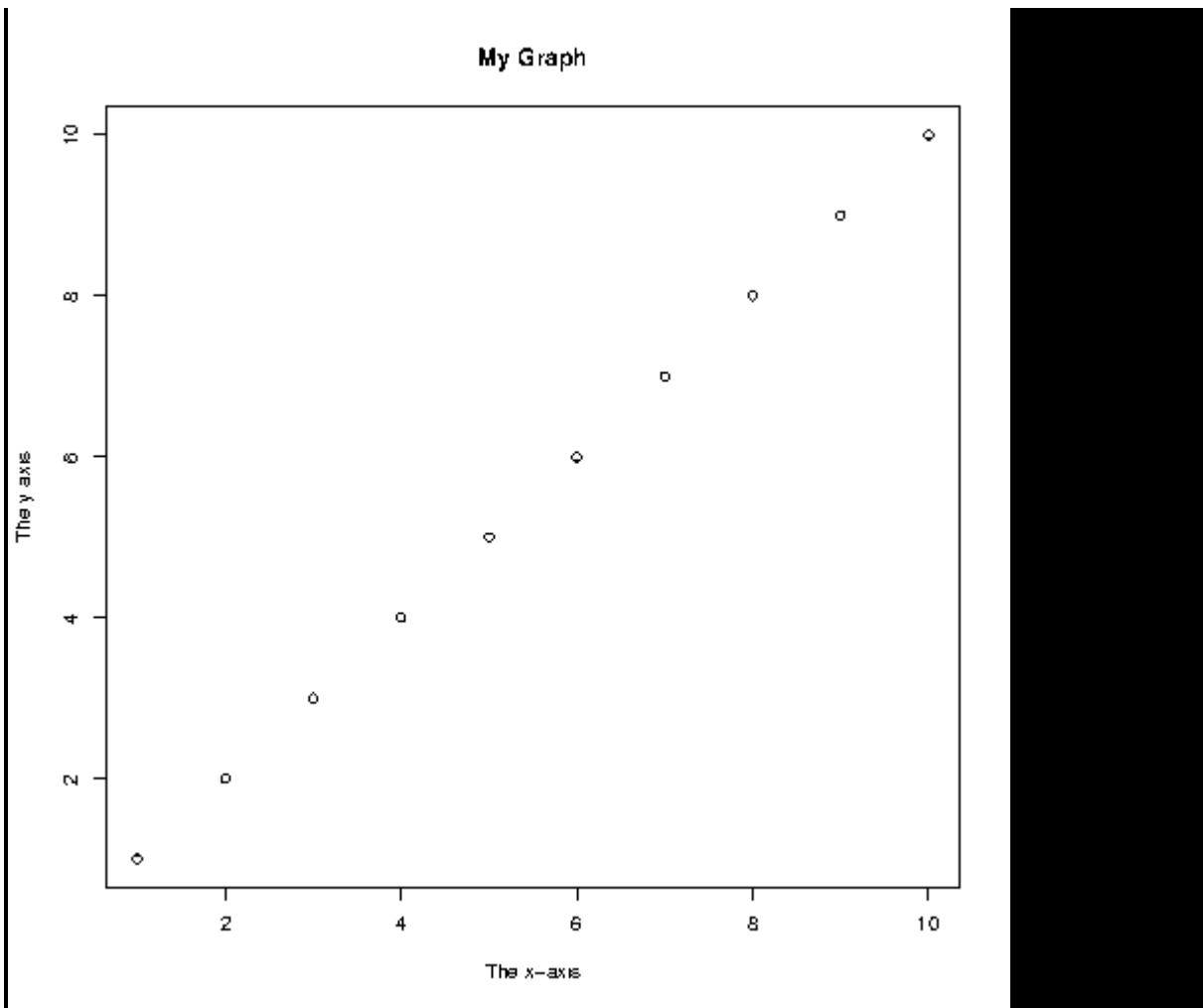
Plot Labels

The `plot()` function also accept other parameters, such as `main`, `xlab` and `ylab` if you want to customize the graph with a main title and different labels for the x and y-axis:

Example

```
plot(1:10, main="My Graph", xlab="The x-axis", ylab="The y axis")
```

Result:



Graph Appearance

There are many other parameters you can use to change the appearance of the points.

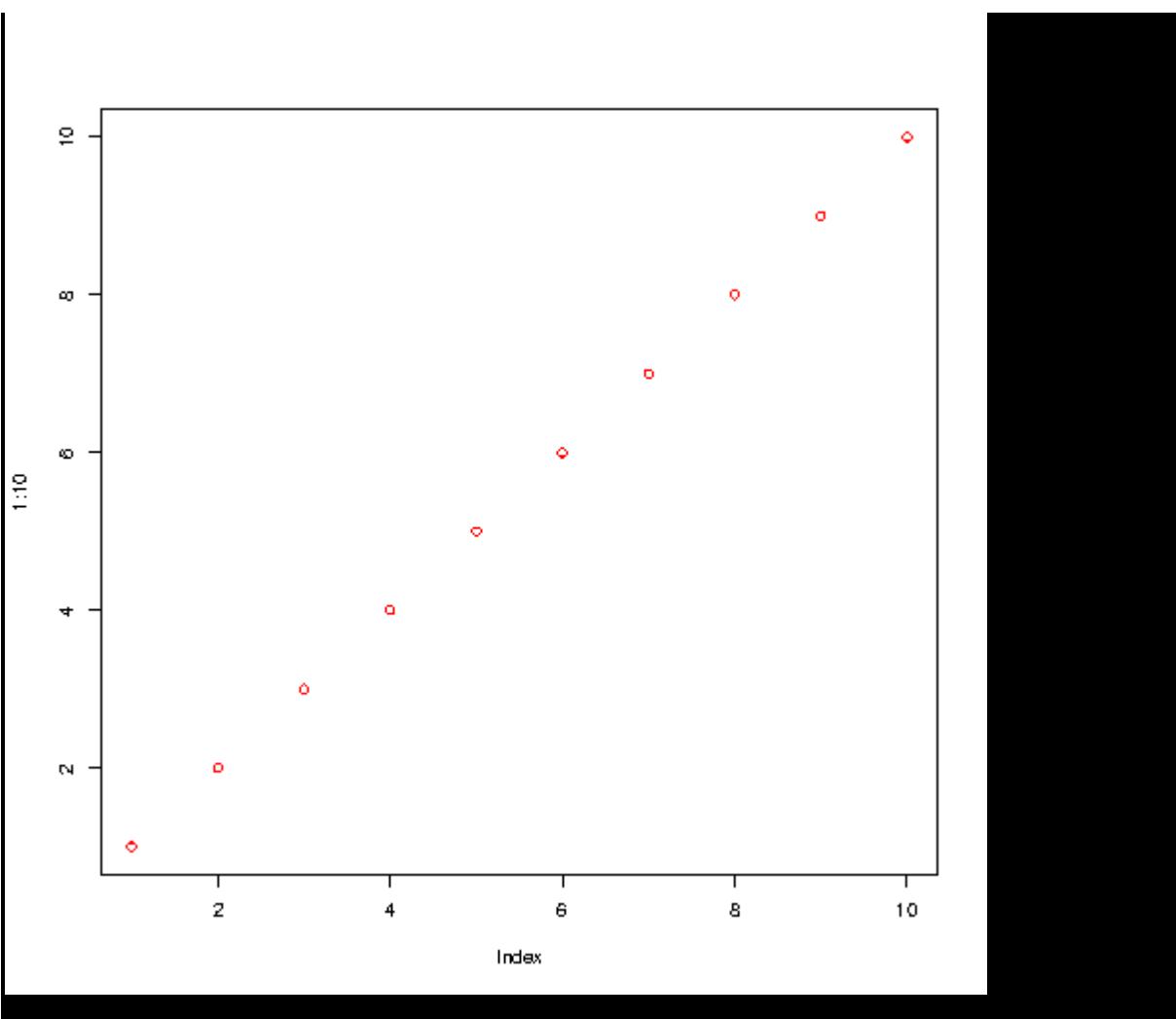
Colors

Use `col="color"` to add a color to the points:

Example

```
plot(1:10, col="red")
```

Result:



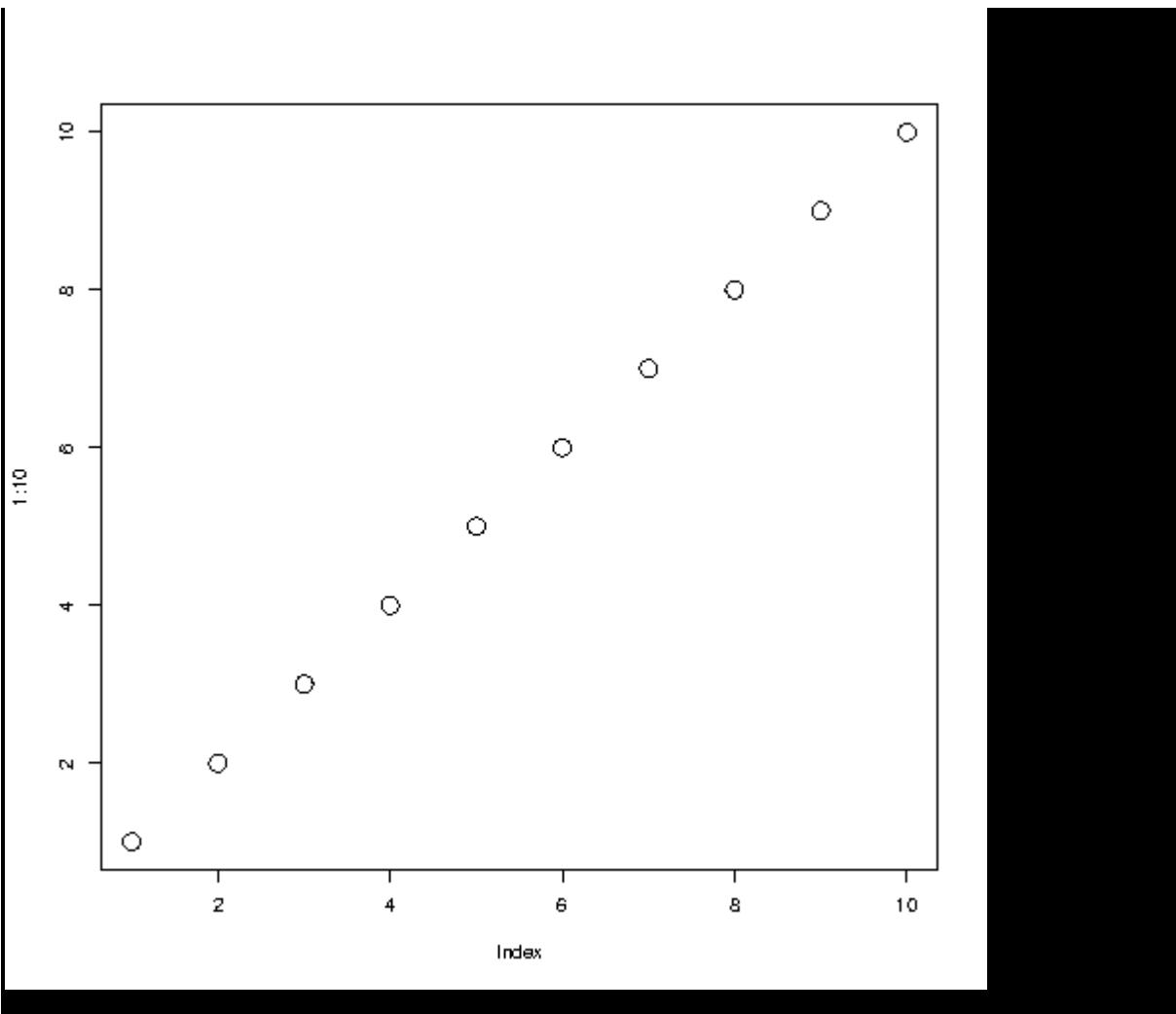
Size

Use `cex=number` to change the size of the points (`1` is default, while `0.5` means 50% smaller, and `2` means 100% larger):

Example

```
plot(1:10, cex=2)
```

Result:



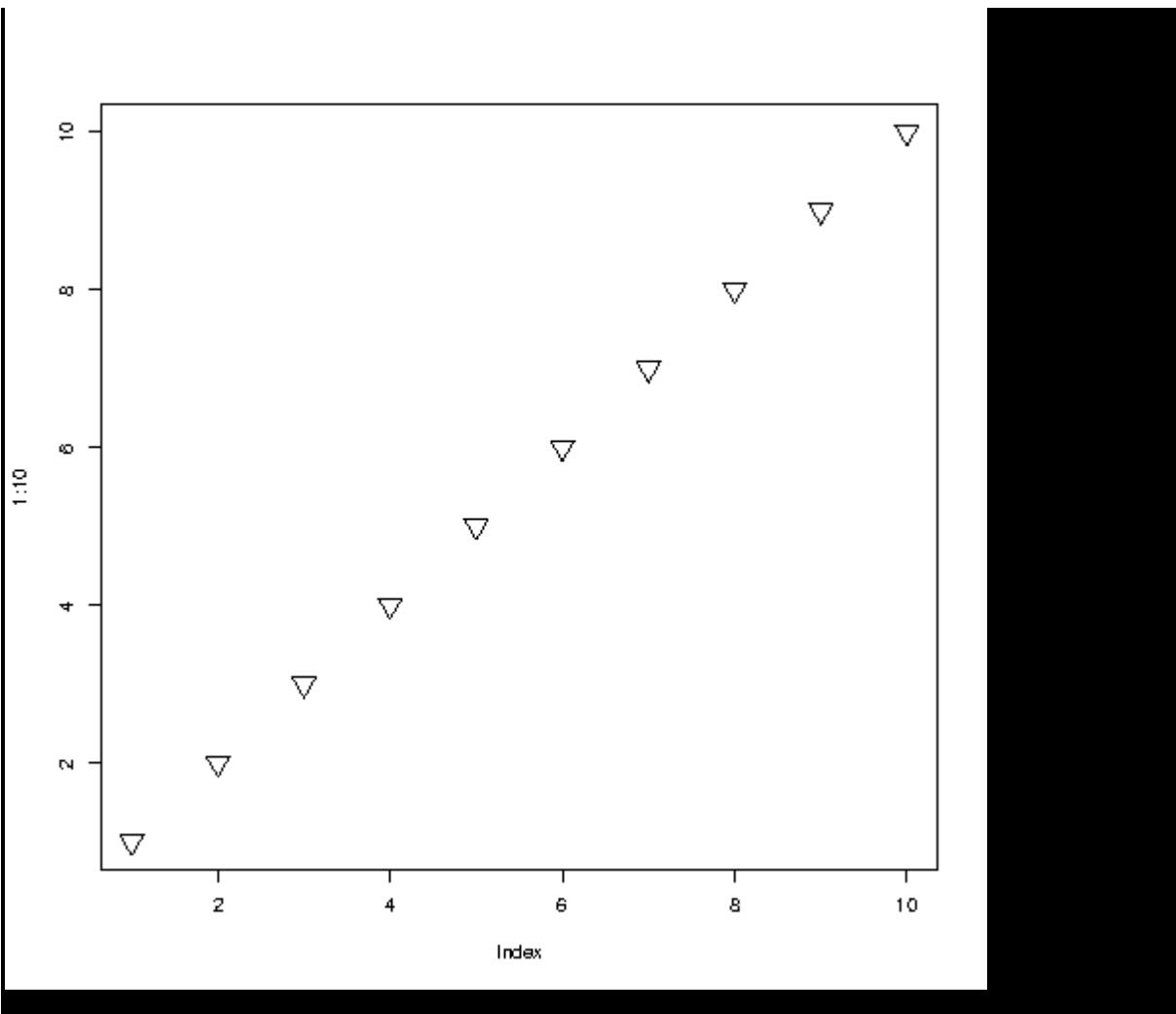
Point Shape

Use `pch` with a value from 0 to 25 to change the point shape format:

Example

```
plot(1:10, pch=25, cex=2)
```

Result:



The values of the `pch` parameter ranges from 0 to 25, which means that we can choose up to 26 different types of point shapes:

R Line

Line Graphs

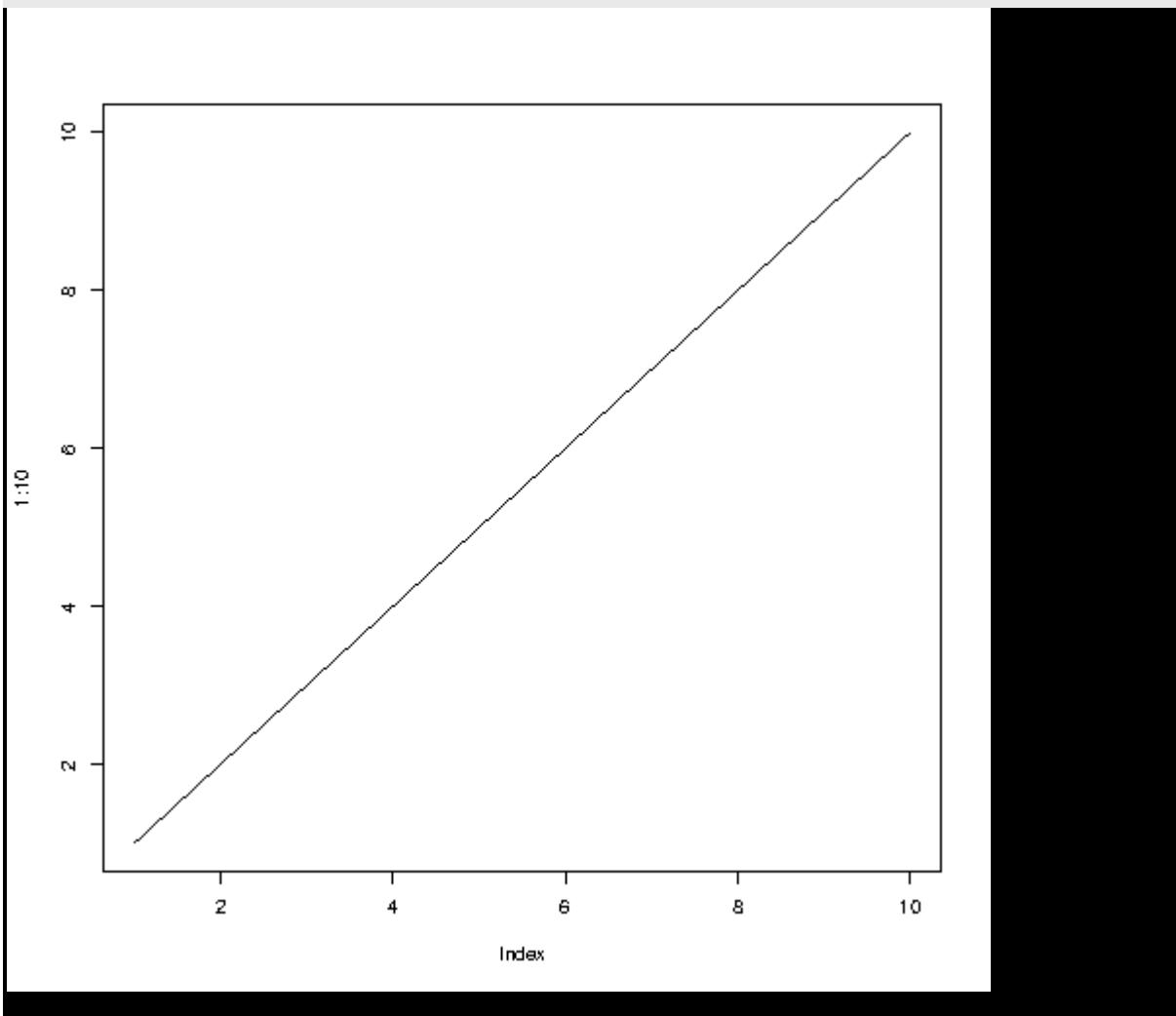
A line graph has a line that connects all the points in a diagram.

To create a line, use the `plot()` function and add the `type` parameter with a value of "`l`":

Example

```
plot(1:10, type="l")
```

Result:



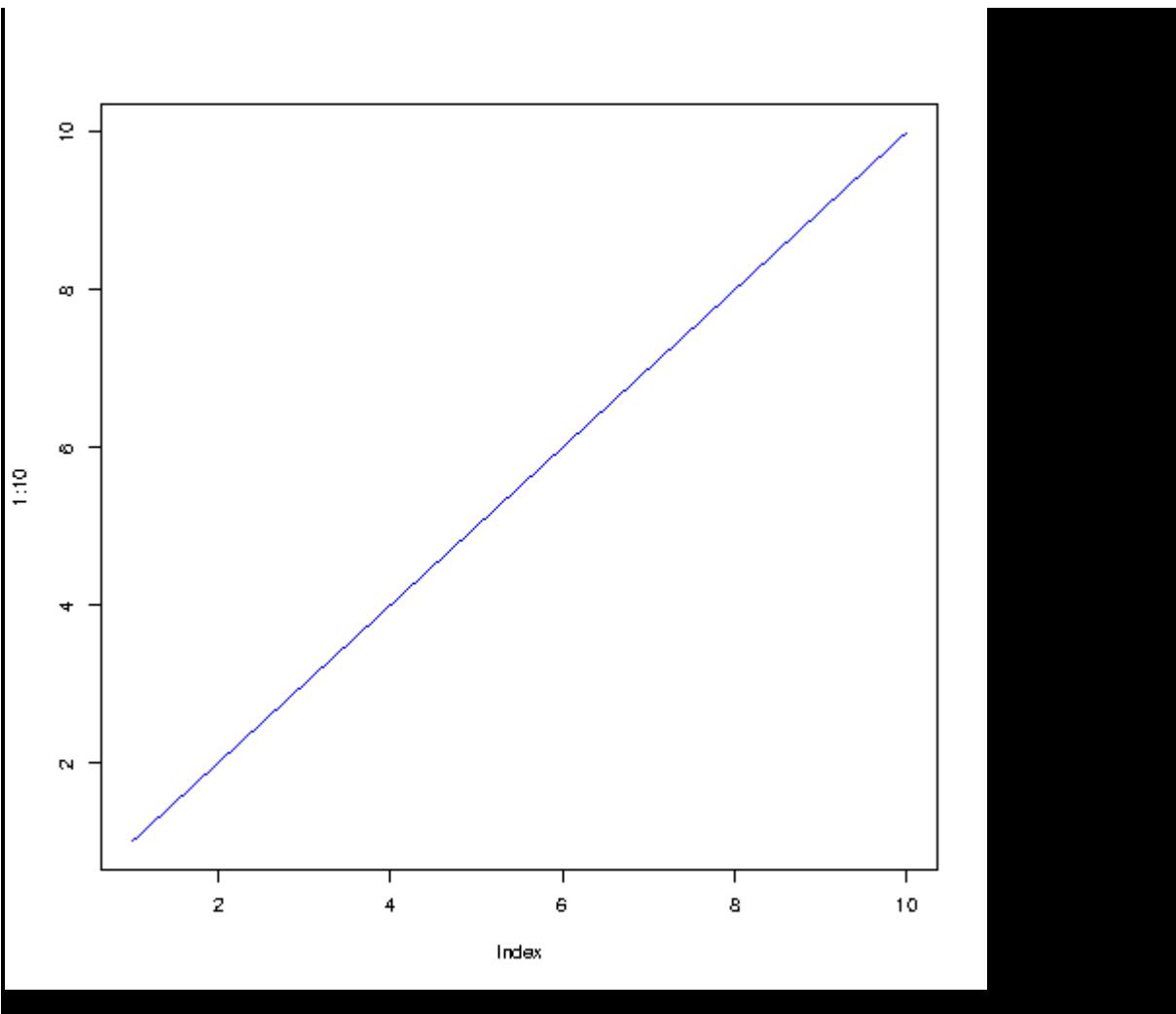
Line Color

The line color is black by default. To change the color, use the `col` parameter:

Example

```
plot(1:10, type="l", col="blue")
```

Result:



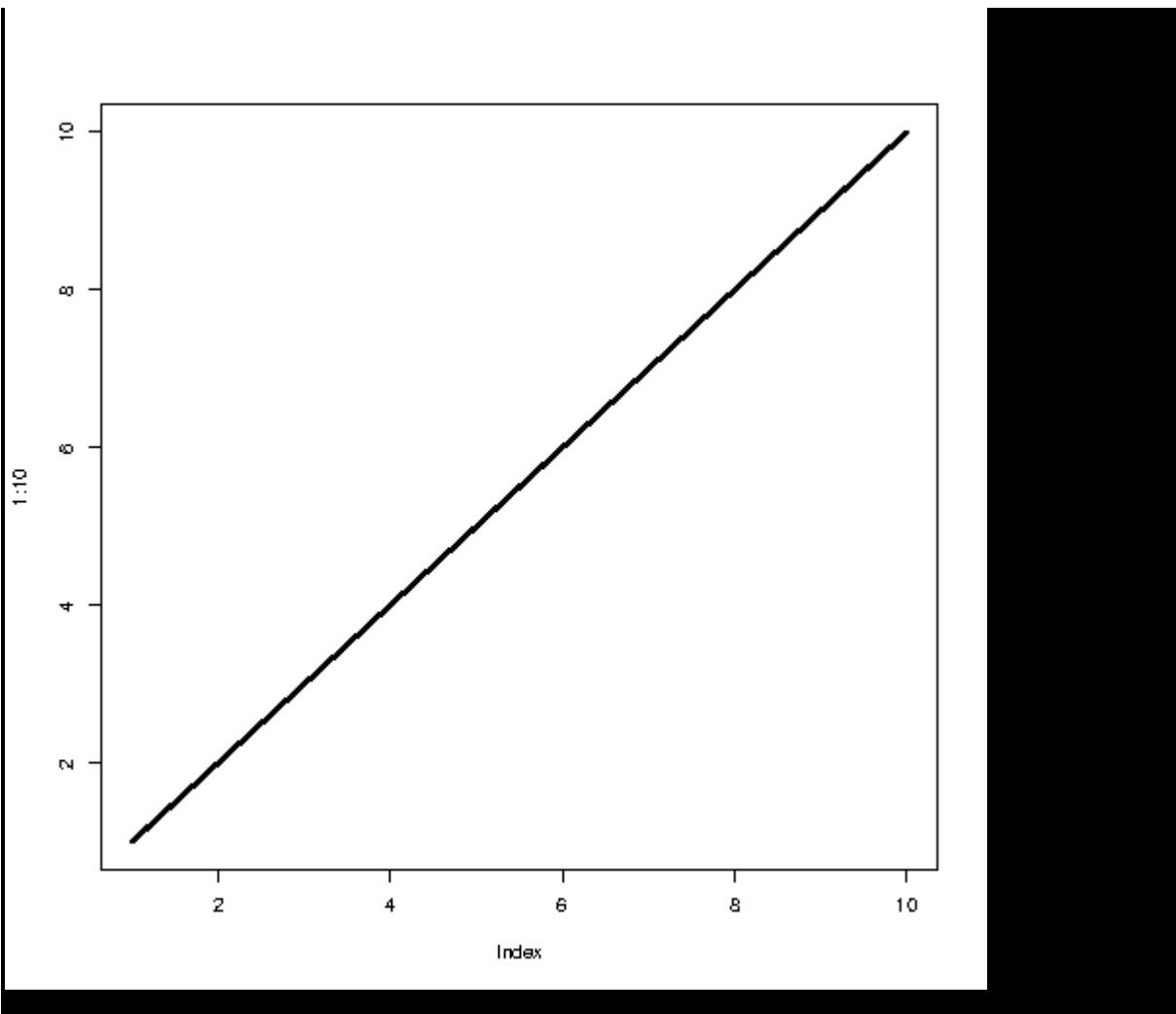
Line Width

To change the width of the line, use the `lwd` parameter (`1` is default, while `0.5` means 50% smaller, and `2` means 100% larger):

Example

```
plot(1:10, type="l", lwd=2)
```

Result:



Line Styles

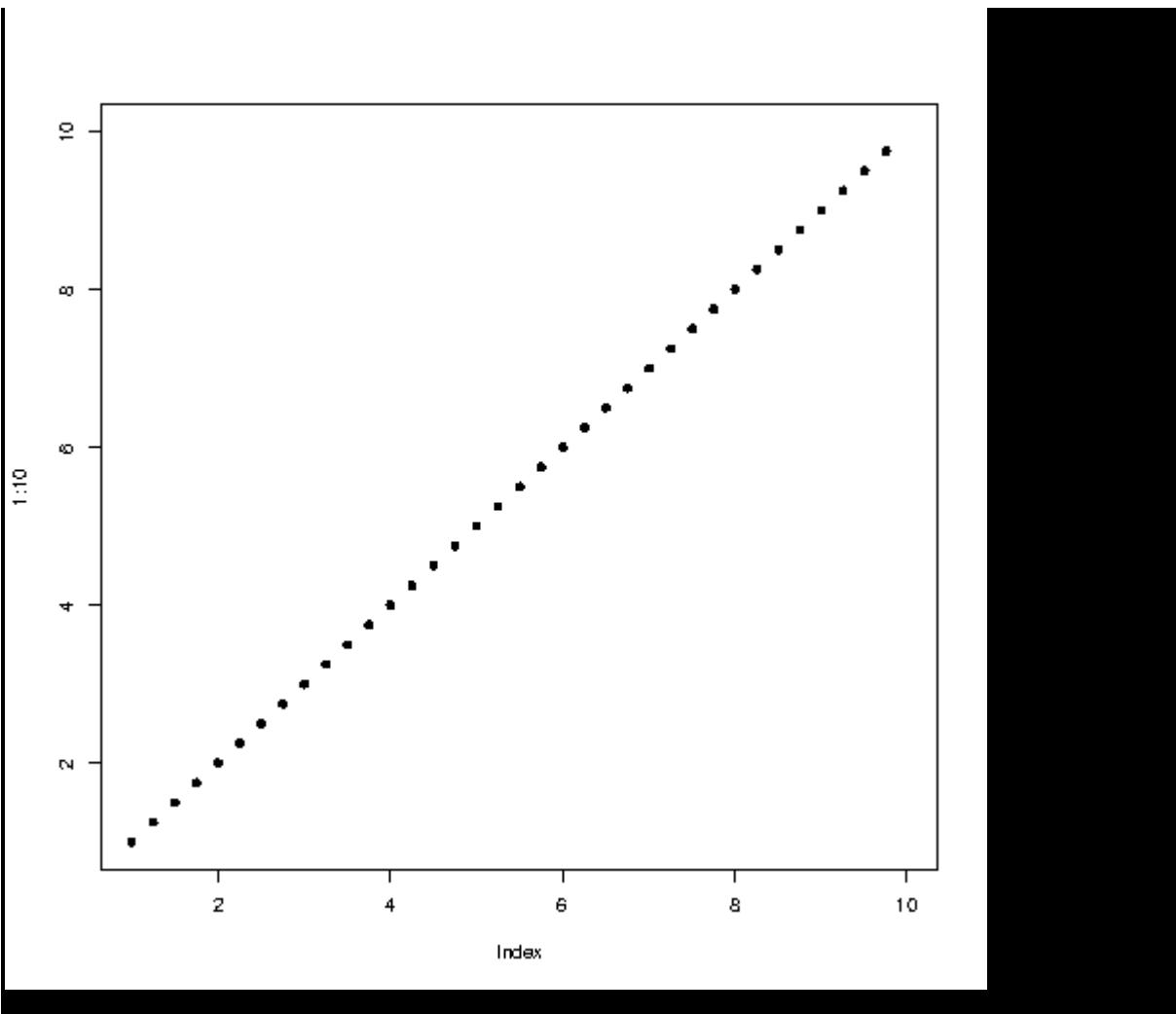
The line is solid by default. Use the `lty` parameter with a value from **0 to 6** to specify the line format.

For example, `lty=3` will display a dotted line instead of a solid line:

Example

```
plot(1:10, type="l", lwd=5, lty=3)
```

Result:



Available parameter values for `lty`:

- `0` removes the line
- `1` displays a solid line
- `2` displays a dashed line
- `3` displays a dotted line
- `4` displays a "dot dashed" line
- `5` displays a "long dashed" line
- `6` displays a "two dashed" line

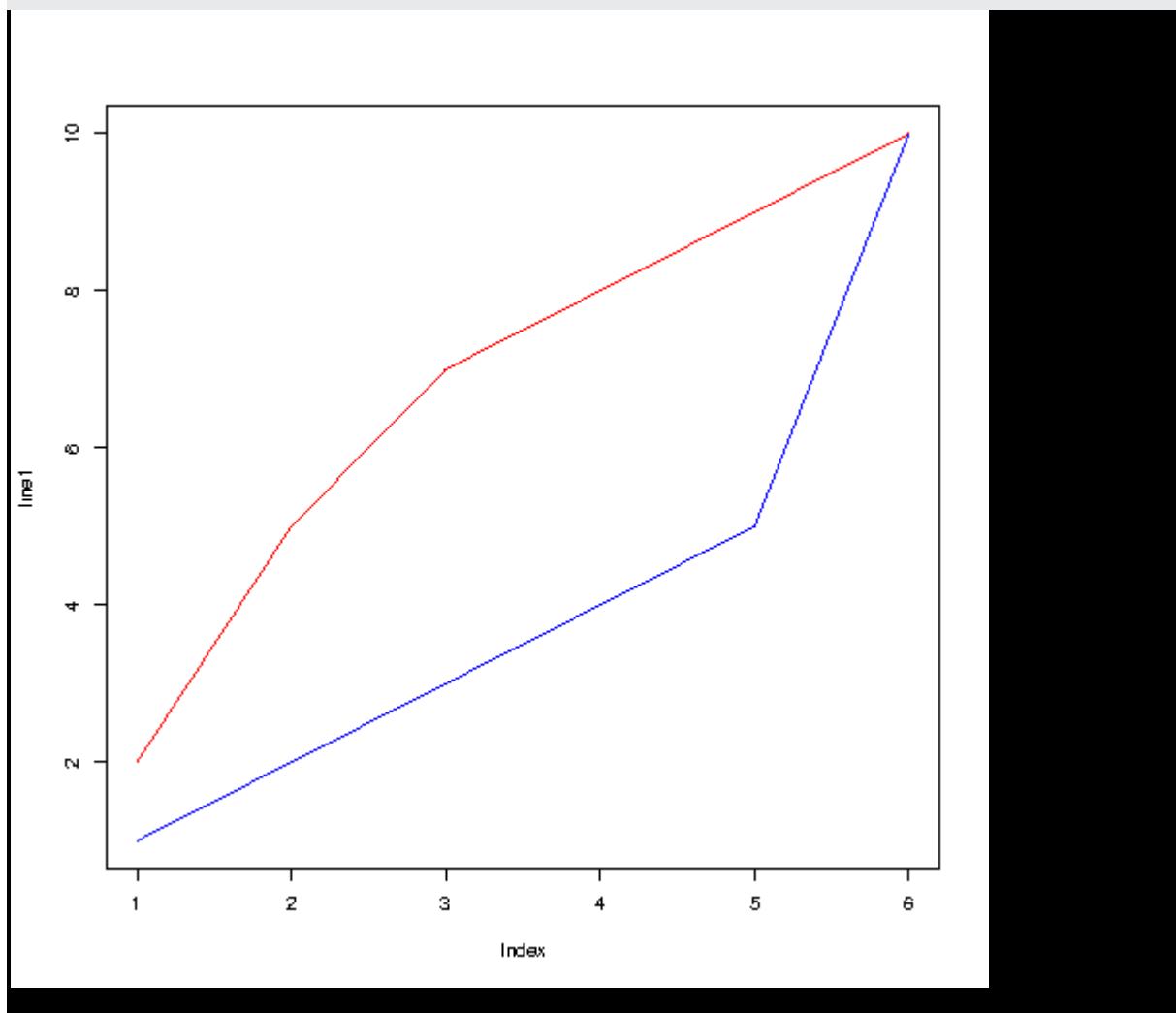
Multiple Lines

To display more than one line in a graph, use the `plot()` function together with the `lines()` function:

Example

```
line1 <- c(1,2,3,4,5,10)  
line2 <- c(2,5,7,8,9,10)  
  
plot(line1, type = "l", col = "blue")  
lines(line2, type="l", col = "red")
```

Result:



R Scatter Plot

Scatter Plots

You learned from the [Plot chapter](#) that the `plot()` function is used to plot numbers against each other.

A "scatter plot" is a type of plot used to display the relationship between two numerical variables, and plots one dot for each observation.

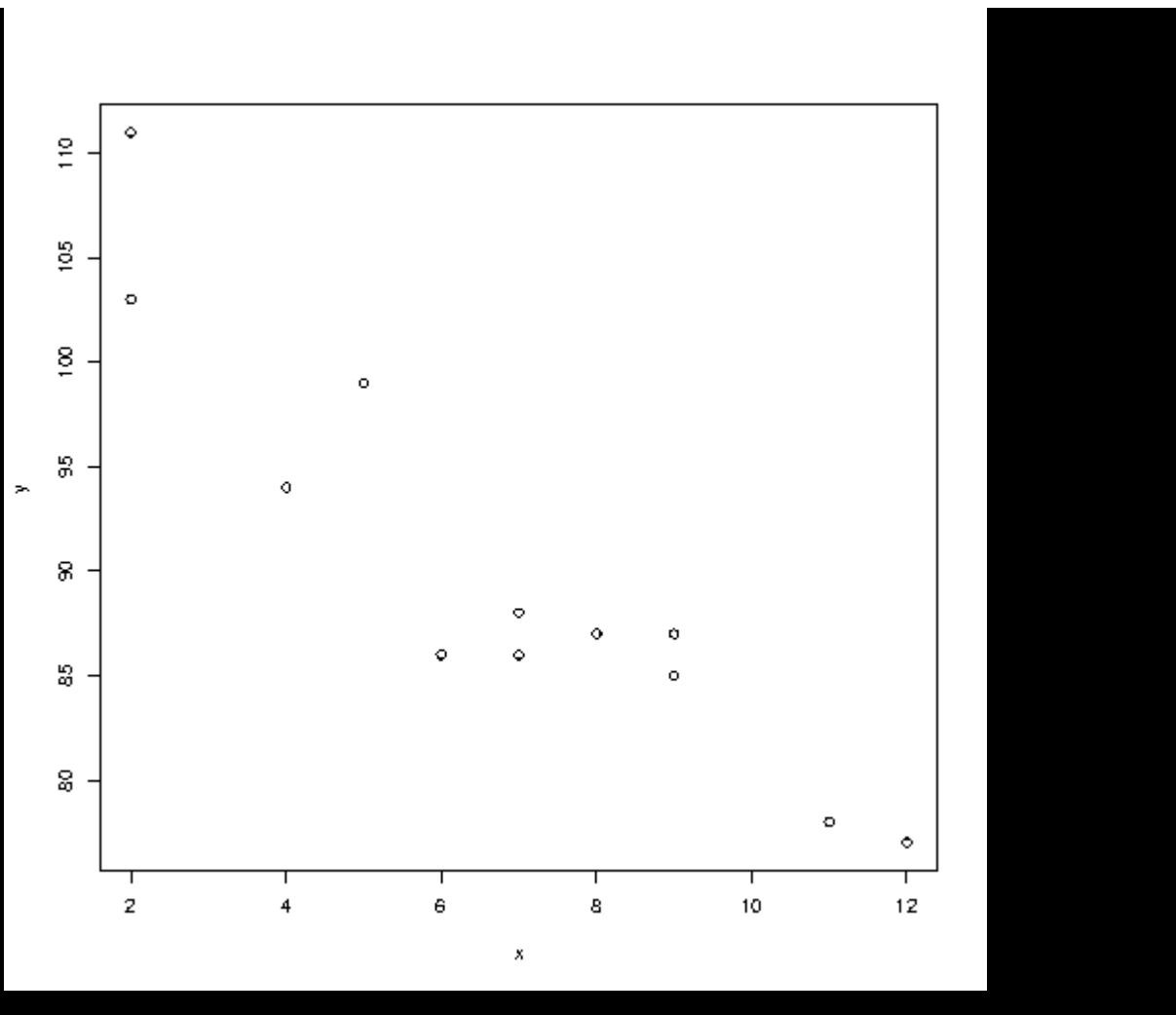
It needs two vectors of same length, one for the x-axis (horizontal) and one for the y-axis (vertical):

Example

```
x <- c(5,7,8,7,2,2,9,4,11,12,9,6)
y <- c(99,86,87,88,111,103,87,94,78,77,85,86)

plot(x, y)
```

Result:



The observation in the example above should show the result of 12 cars passing by.

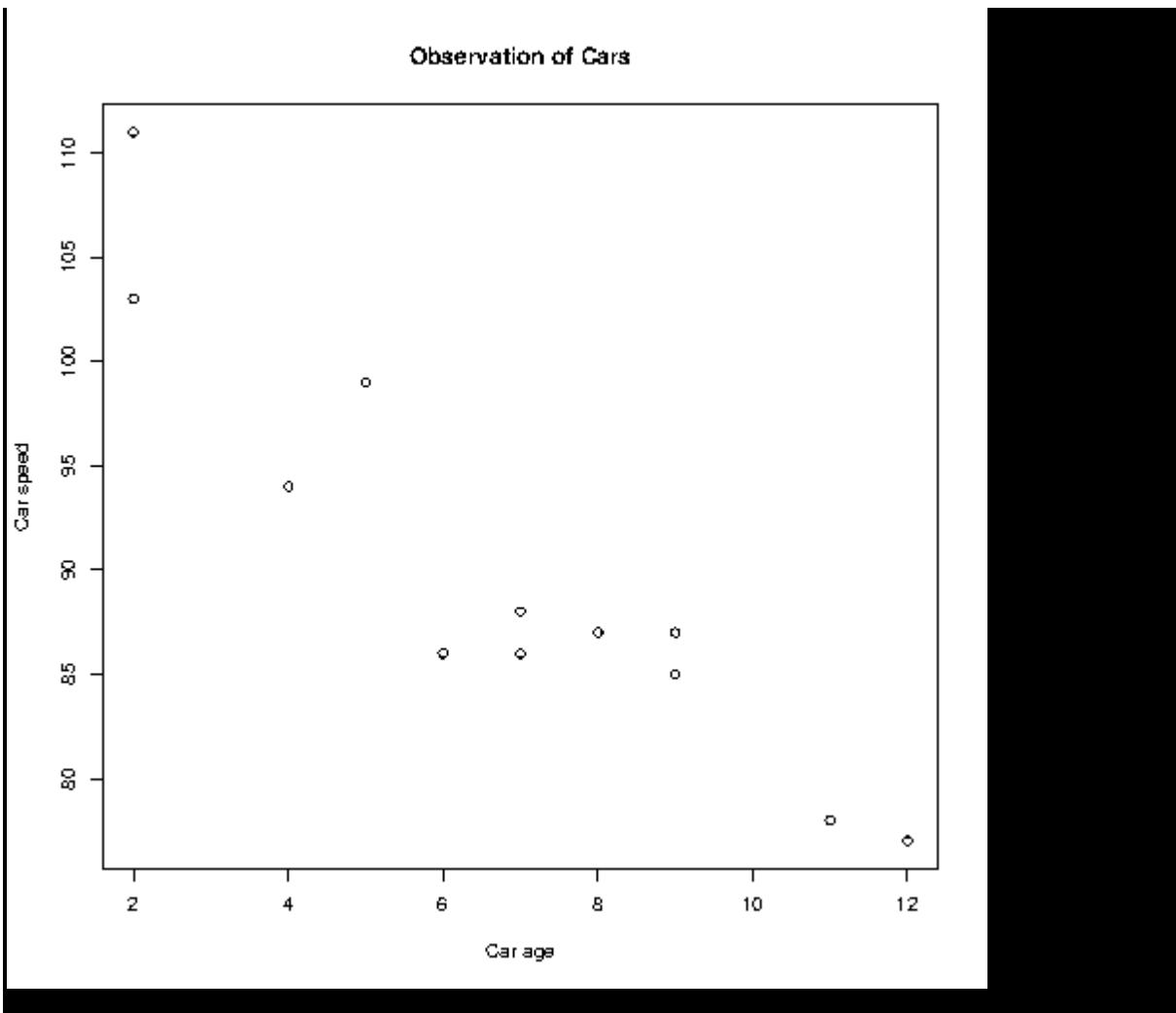
That might not be clear for someone who sees the graph for the first time, so let's add a header and different labels to describe the scatter plot better:

Example

```
x <- c(5,7,8,7,2,2,9,4,11,12,9,6)
y <- c(99,86,87,88,111,103,87,94,78,77,85,86)

plot(x, y, main="Observation of Cars", xlab="Car age", ylab="Car
speed")
```

Result:



To recap, the observation in the example above is the result of 12 cars passing by.

The **x-axis** shows how old the car is.

The **y-axis** shows the speed of the car when it passes.

Are there any relationships between the observations?

It seems that the newer the car, the faster it drives, but that could be a coincidence, after all we only registered 12 cars.

Compare Plots

In the example above, there seems to be a relationship between the car speed and age, but what if we plot the observations from another day as well? Will the scatter plot tell us something else?

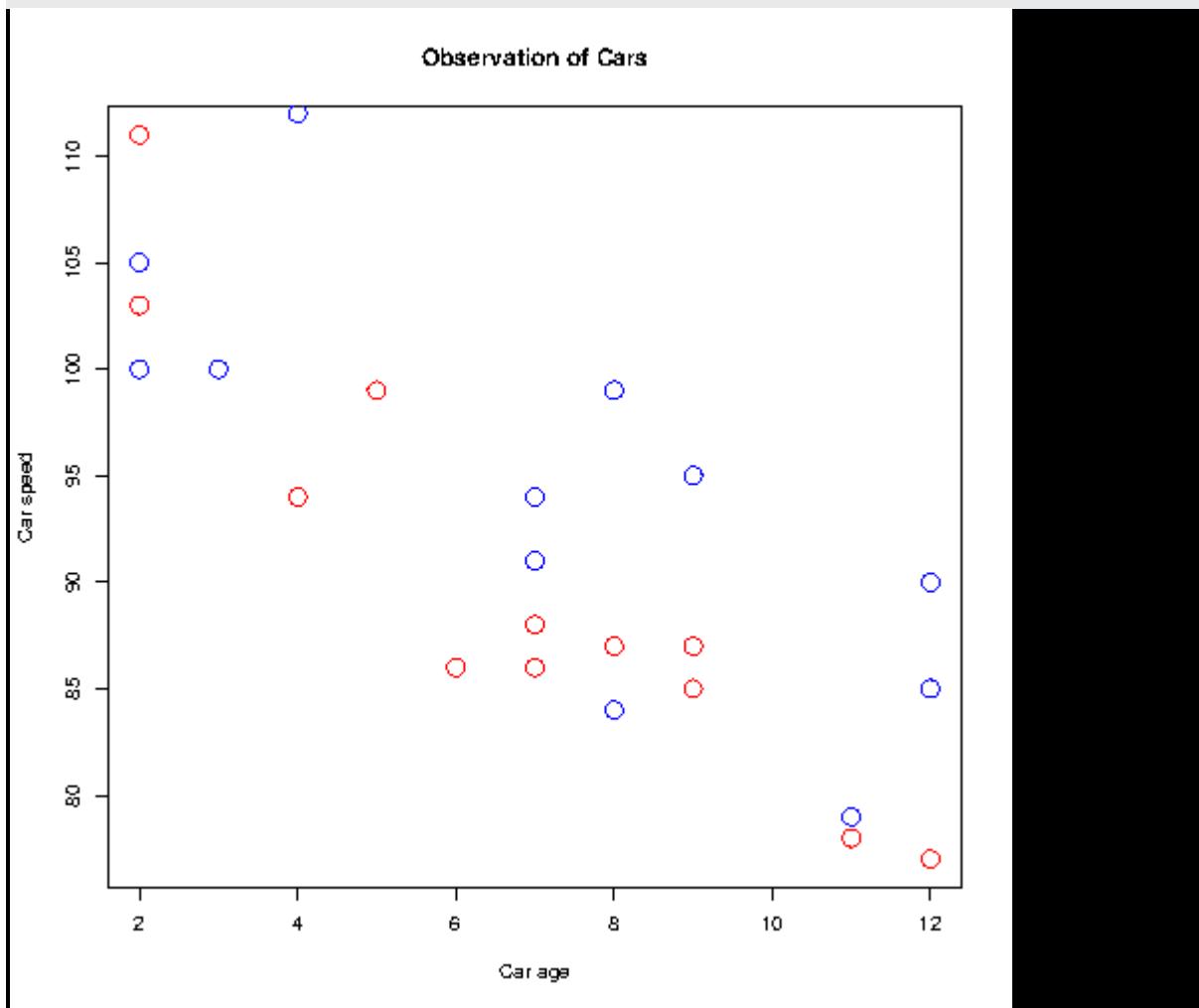
To compare the plot with another plot, use the `points()` function:

Example

Draw two plots on the same figure:

```
# day one, the age and speed of 12 cars:  
x1 <- c(5,7,8,7,2,2,9,4,11,12,9,6)  
y1 <- c(99,86,87,88,111,103,87,94,78,77,85,86)  
  
# day two, the age and speed of 15 cars:  
x2 <- c(2,2,8,1,15,8,12,9,7,3,11,4,7,14,12)  
y2 <- c(100,105,84,105,90,99,90,95,94,100,79,112,91,80,85)  
  
plot(x1, y1, main="Observation of Cars", xlab="Car age", ylab="Car  
speed", col="red", cex=2)  
points(x2, y2, col="blue", cex=2)
```

Result:



Note: To be able to see the difference of the comparison, you must assign different colors to the plots (by using the `col` parameter). Red represents the

values of day 1, while `blue` represents day 2. Note that we have also added the `cex` parameter to increase the size of the dots.

Conclusion of observation: By comparing the two plots, I think it is safe to say that they both gives us the same conclusion: the newer the car, the faster it drives.

R Pie Charts

Pie Charts

A pie chart is a circular graphical view of data.

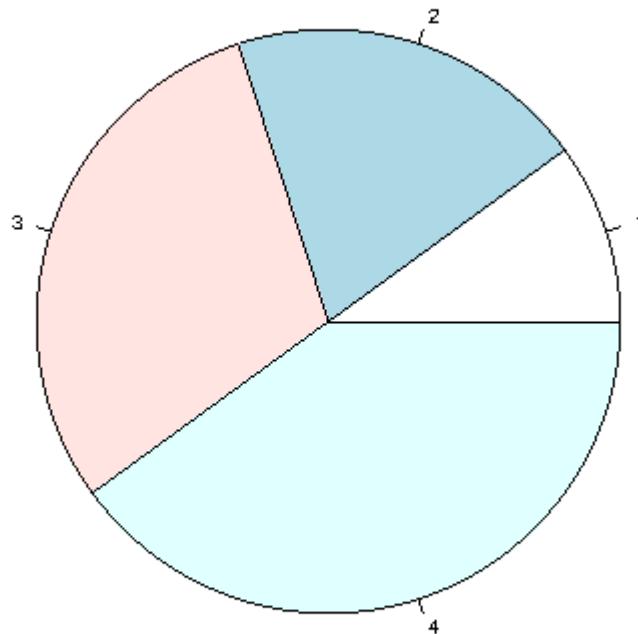
Use the `pie()` function to draw pie charts:

Example

```
# Create a vector of pies
x <- c(10,20,30,40)

# Display the pie chart
pie(x)
```

Result:



Example Explained

As you can see the pie chart draws one pie for each value in the vector (in this case 10, 20, 30, 40).

By default, the plotting of the first pie starts from the x-axis and move **counterclockwise**.

Note: The size of each pie is determined by comparing the value with all the other values, by using this formula:

The value divided by the sum of all values: $x/\text{sum}(x)$

Start Angle

You can change the start angle of the pie chart with the `init.angle` parameter.

The value of `init.angle` is defined with angle in degrees, where default angle is 0.

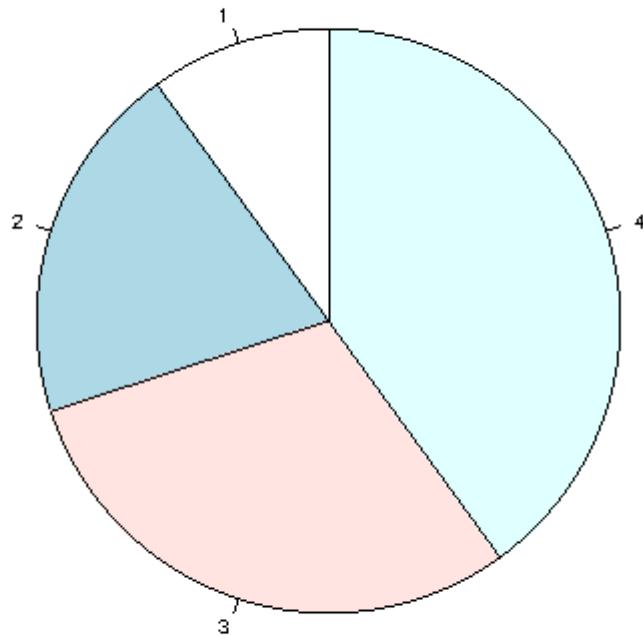
Example

Start the first pie at 90 degrees:

```
# Create a vector of pies
x <- c(10,20,30,40)

# Display the pie chart and start the first pie at 90 degrees
pie(x, init.angle = 90)
```

Result:



Labels and Header

Use the `label` parameter to add a label to the pie chart, and use the `main` parameter to add a header:

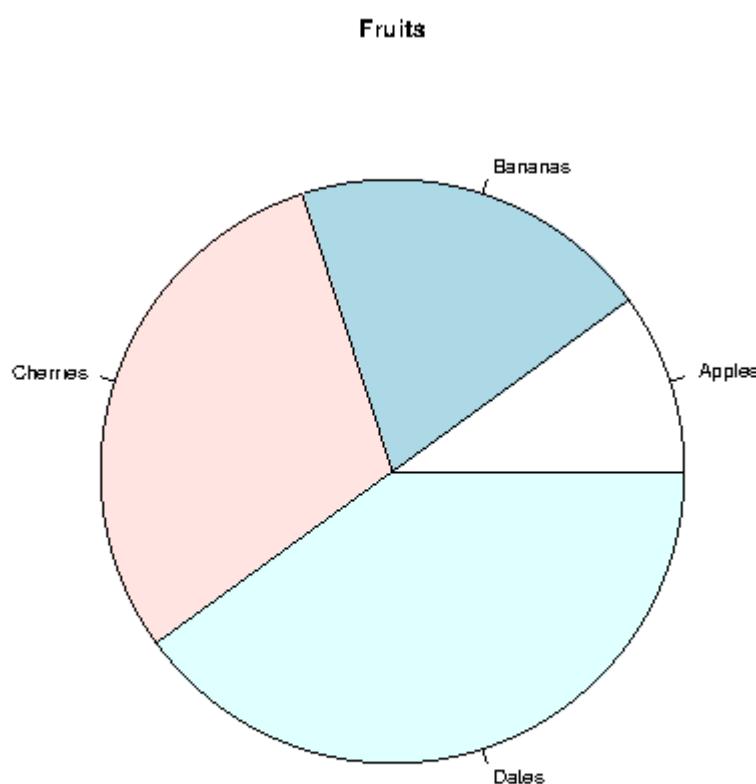
Example

```
# Create a vector of pies
x <- c(10,20,30,40)

# Create a vector of labels
mylabel <- c("Apples", "Bananas", "Cherries", "Dates")

# Display the pie chart with labels
pie(x, label = mylabel, main = "Fruits")
```

Result:



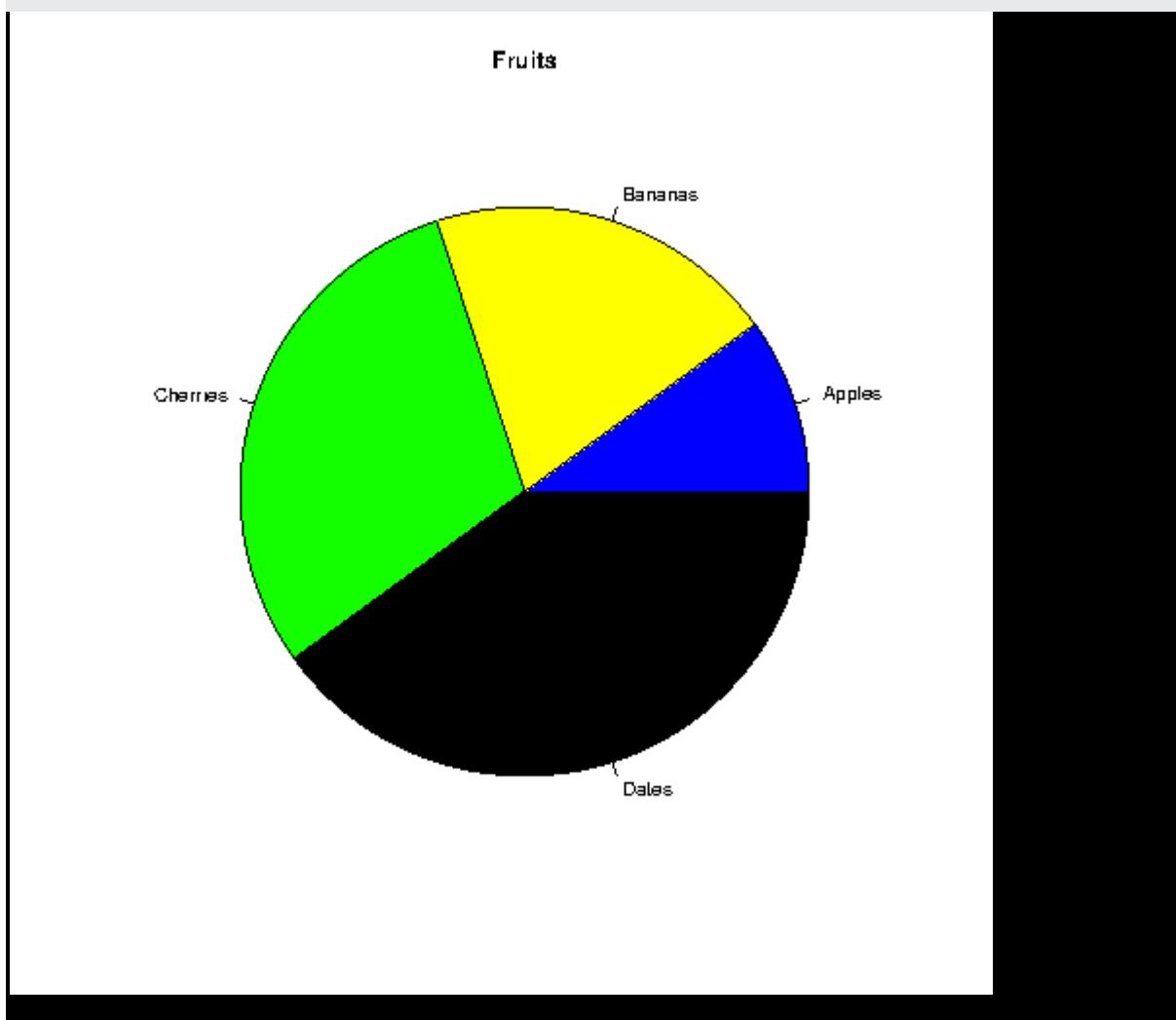
Colors

You can add a color to each pie with the `col` parameter:

Example

```
# Create a vector of colors  
colors <- c("blue", "yellow", "green", "black")  
  
# Display the pie chart with colors  
pie(x, label = mylabel, main = "Fruits", col = colors)
```

Result:



Legend

To add a list of explanation for each pie, use the `legend()` function:

Example

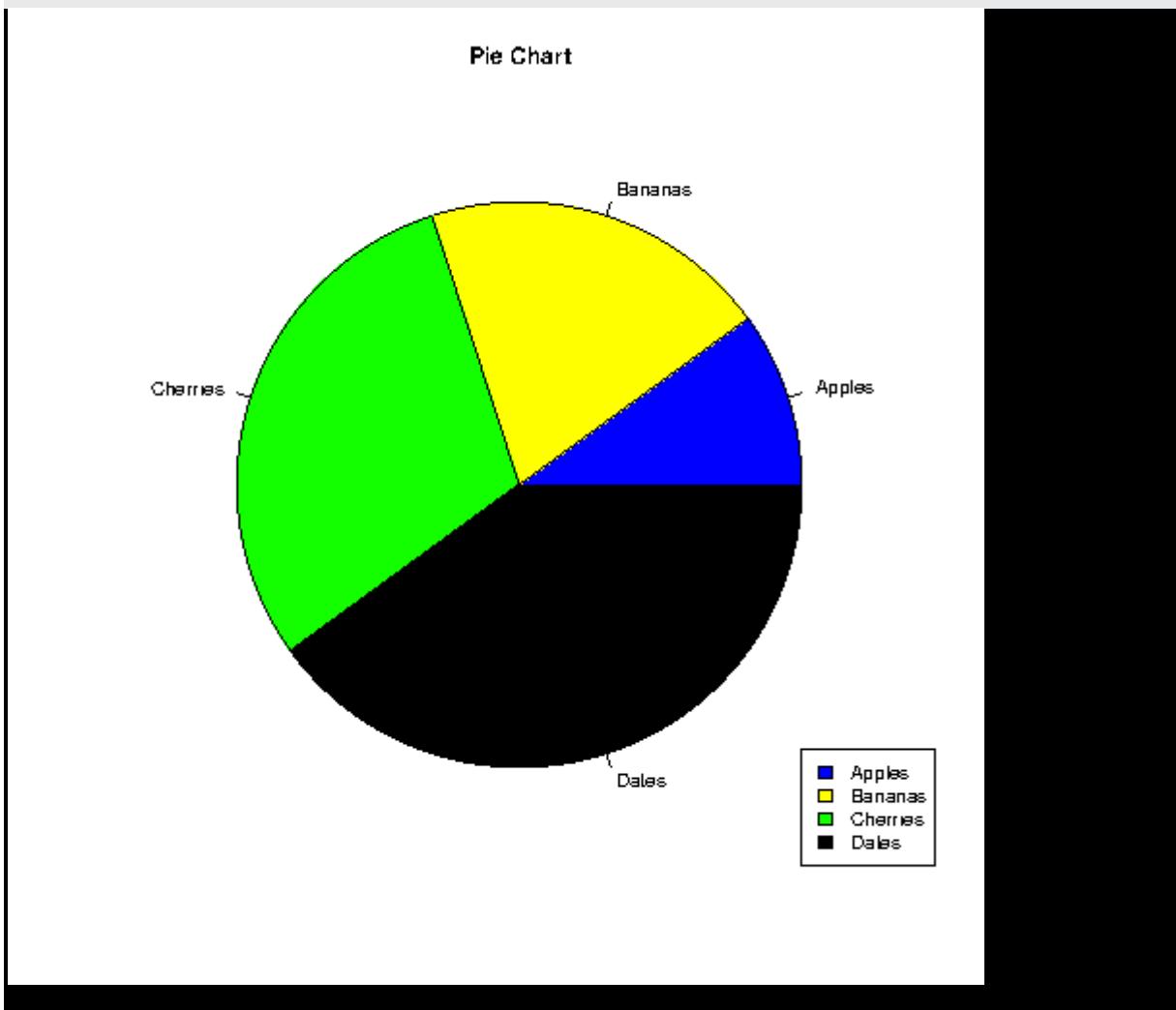
```
# Create a vector of labels  
mylabel <- c("Apples", "Bananas", "Cherries", "Dates")
```

```
# Create a vector of colors
colors <- c("blue", "yellow", "green", "black")

# Display the pie chart with colors
pie(x, label = mylabel, main = "Pie Chart", col = colors)

# Display the explanation box
legend("bottomright", mylabel, fill = colors)
```

Result:



The legend can be positioned as either:

`bottomright, bottom, bottomleft, left, topleft, top, topright, right, center`

R Bar Charts

Bar Charts

A bar chart uses rectangular bars to visualize data. Bar charts can be displayed horizontally or vertically. The height or length of the bars are proportional to the values they represent.

Use the `barplot()` function to draw a vertical bar chart:

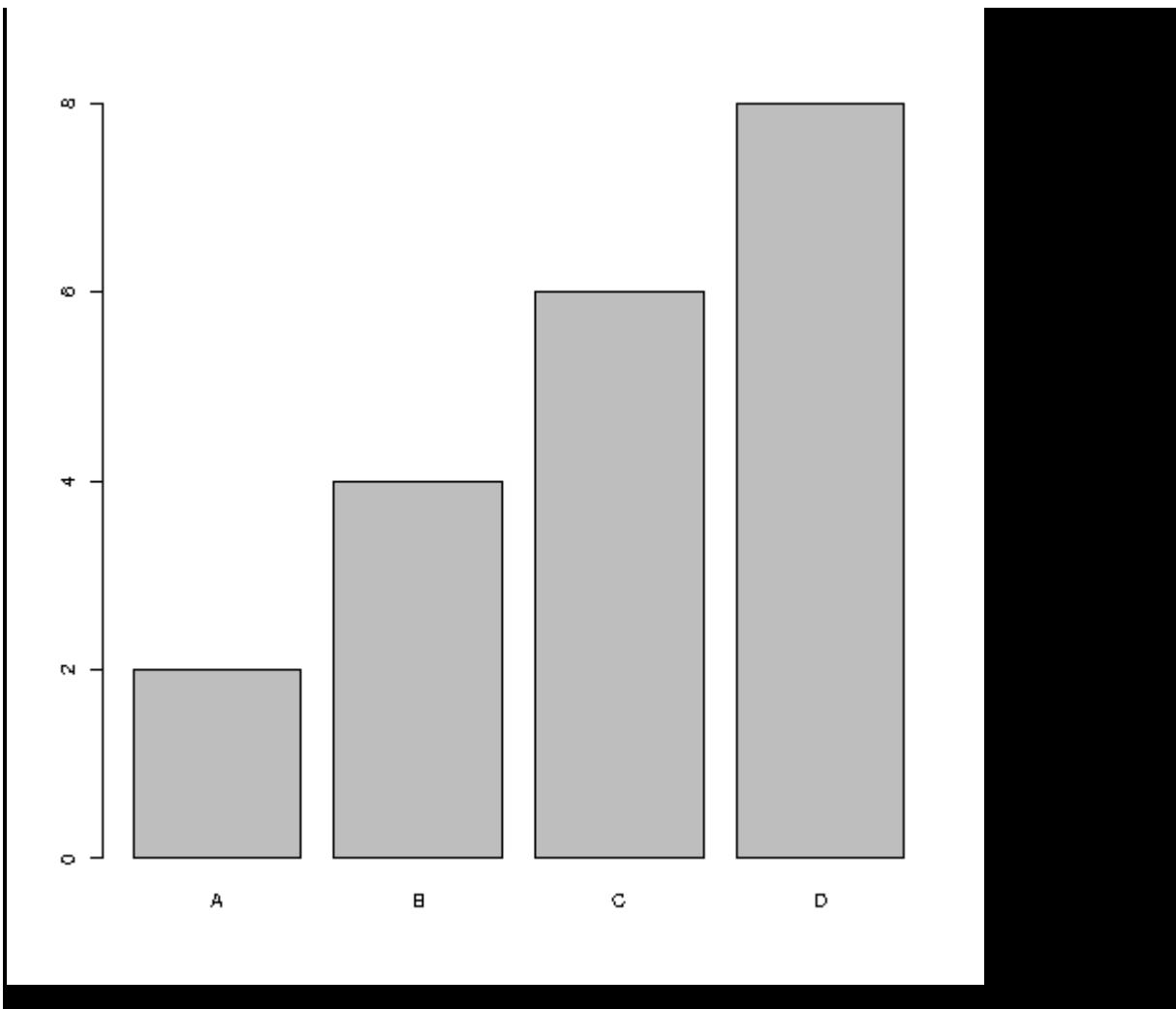
Example

```
# x-axis values
x <- c("A", "B", "C", "D")

# y-axis values
y <- c(2, 4, 6, 8)

barplot(y, names.arg = x)
```

Result:



Example Explained

- The `x` variable represents values in the x-axis (A,B,C,D)
- The `y` variable represents values in the y-axis (2,4,6,8)
- Then we use the `barplot()` function to create a bar chart of the values
- `names.arg` defines the names of each observation in the x-axis

Bar Color

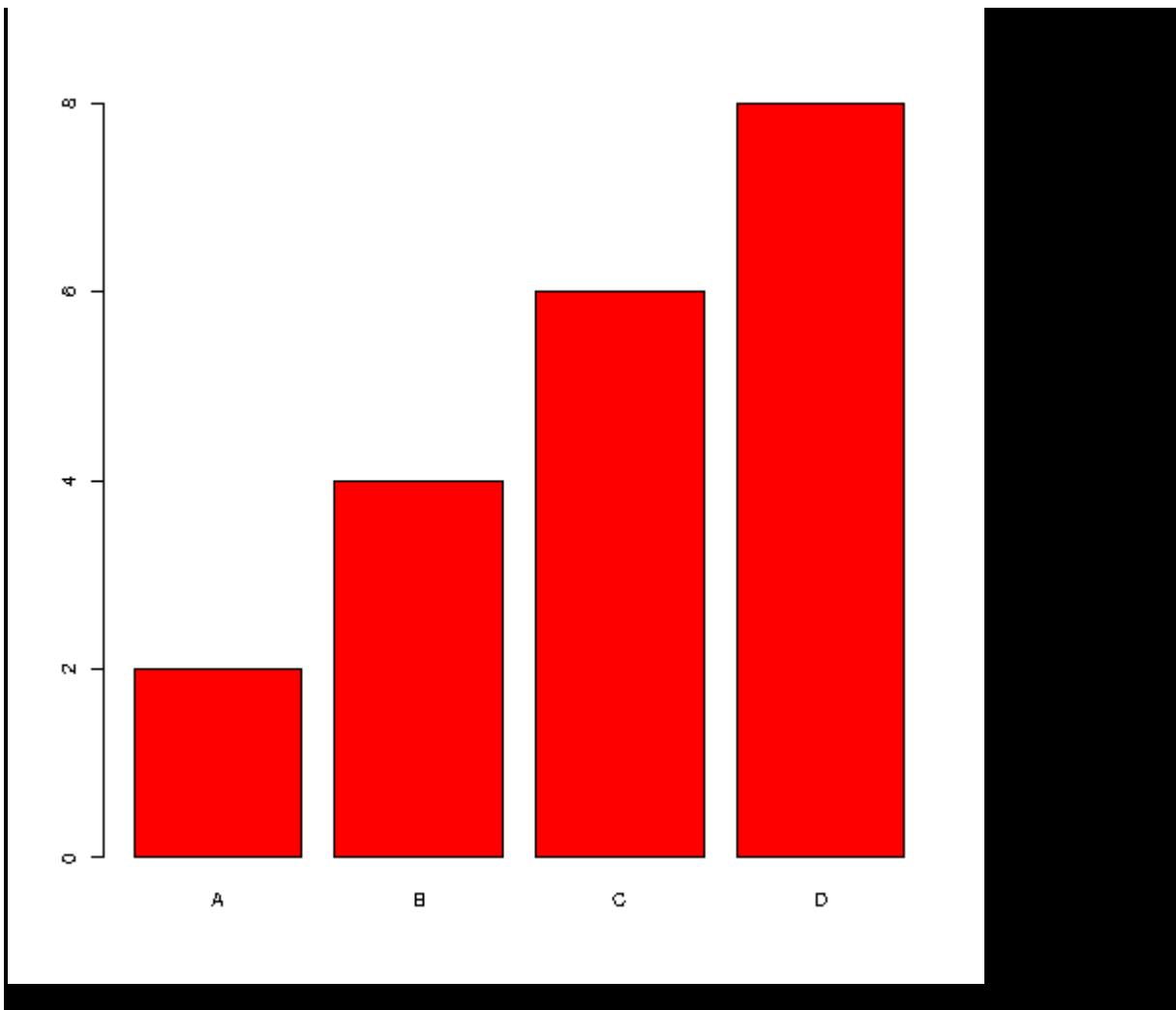
Use the `col` parameter to change the color of the bars:

Example

```
x <- c("A", "B", "C", "D")
y <- c(2, 4, 6, 8)

barplot(y, names.arg = x, col = "red")
```

Result:



Density / Bar Texture

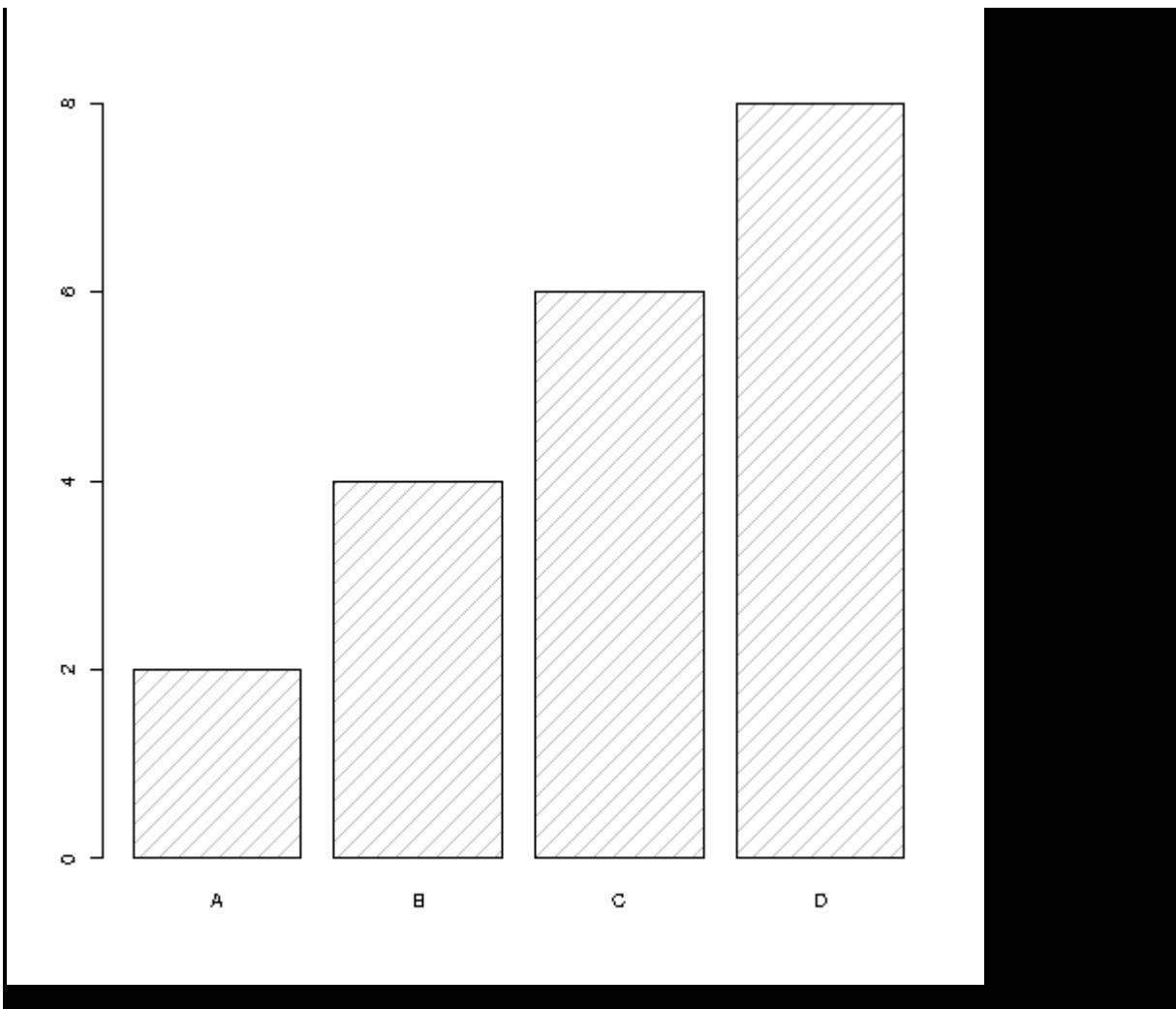
To change the bar texture, use the `density` parameter:

Example

```
x <- c("A", "B", "C", "D")
y <- c(2, 4, 6, 8)

barplot(y, names.arg = x, density = 10)
```

Result:



Bar Width

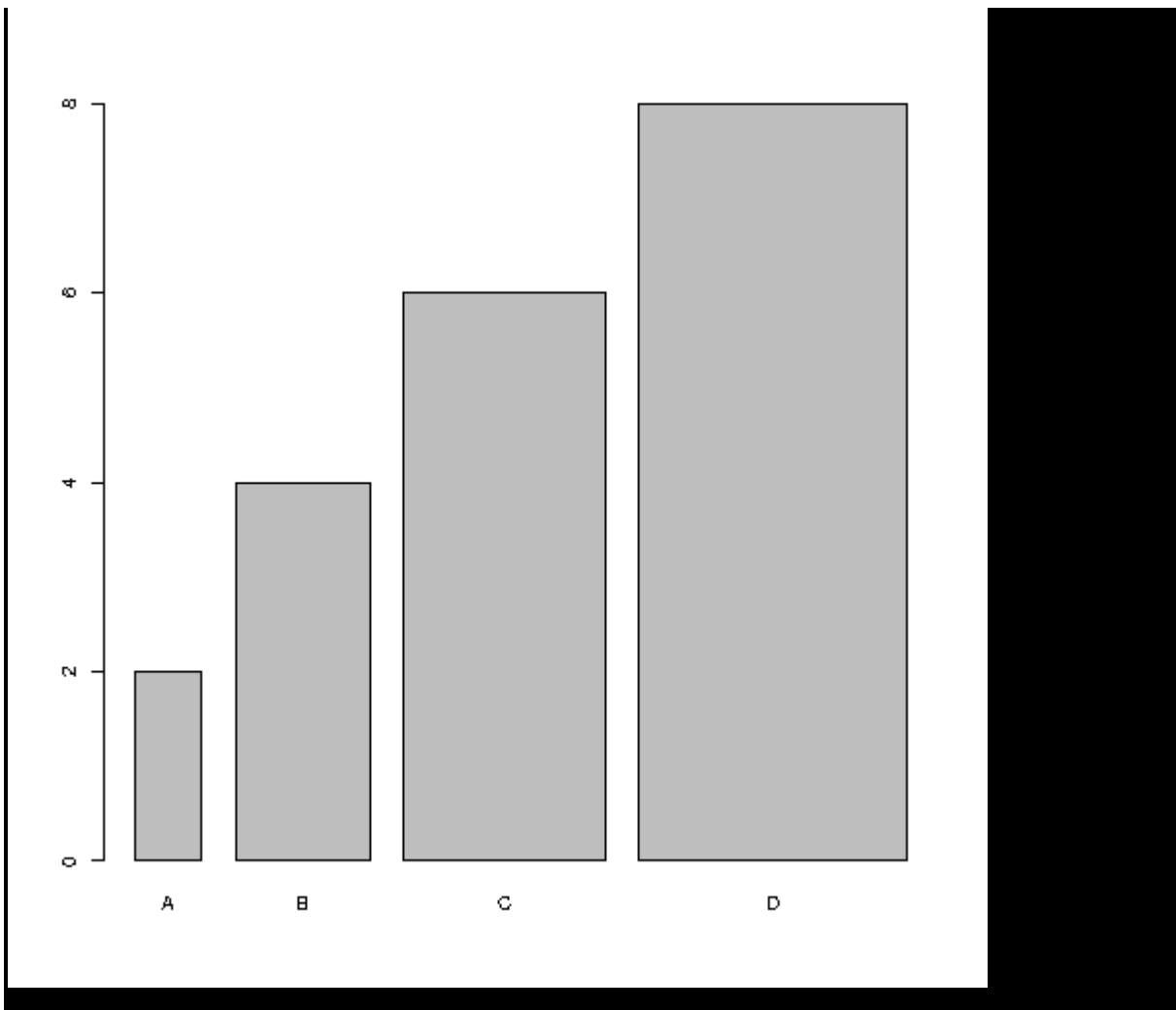
Use the `width` parameter to change the width of the bars:

Example

```
x <- c("A", "B", "C", "D")
y <- c(2, 4, 6, 8)

barplot(y, names.arg = x, width = c(1,2,3,4))
```

Result:



Horizontal Bars

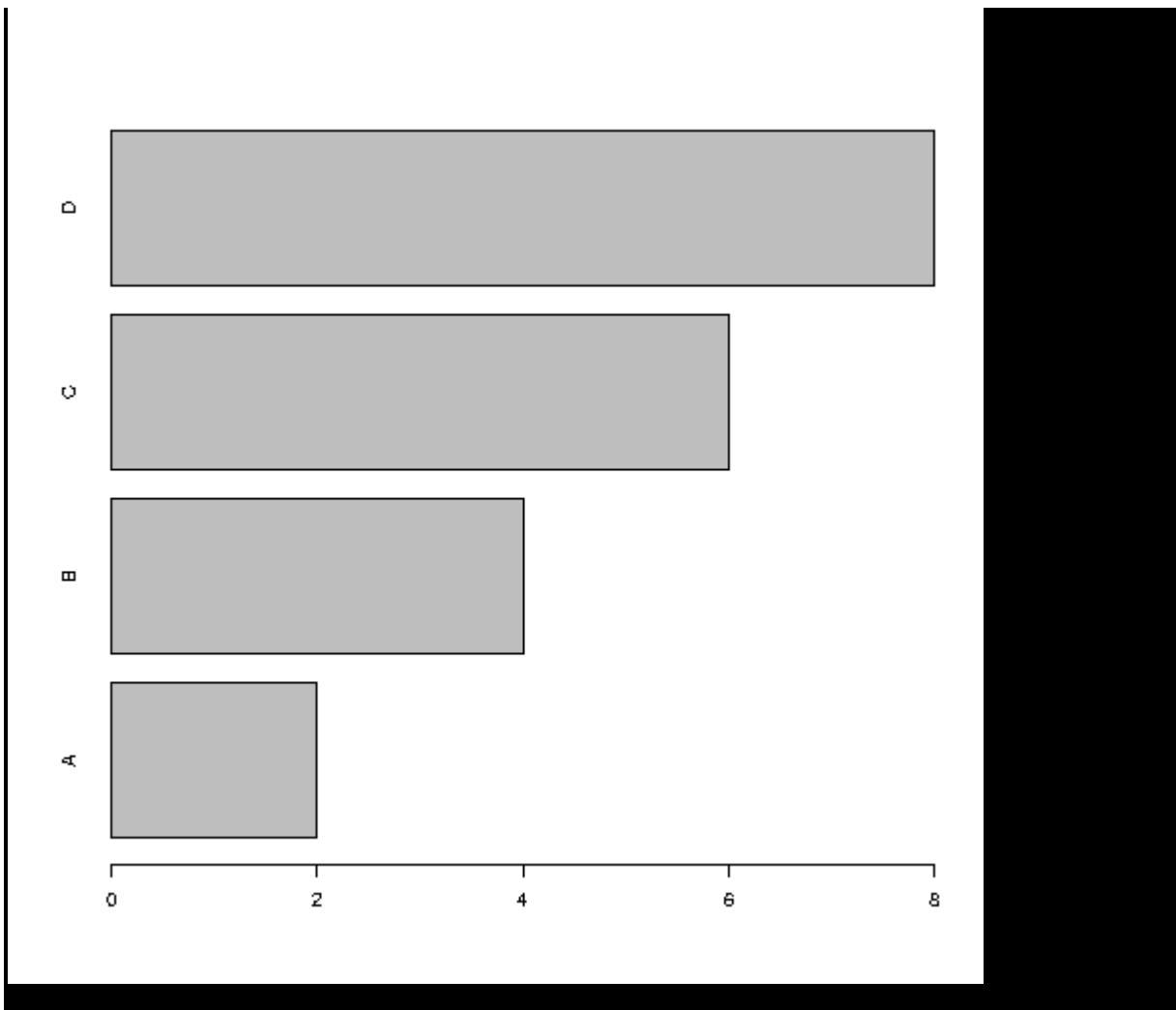
If you want the bars to be displayed horizontally instead of vertically, use `horiz=TRUE`:

Example

```
x <- c("A", "B", "C", "D")
y <- c(2, 4, 6, 8)

barplot(y, names.arg = x, horiz = TRUE)
```

Result:



R Statistics

R Statistics

Statistics Introduction

Statistics is the science of analyzing, reviewing and conclude data.

Some basic statistical numbers include:

- Mean, median and mode
 - Minimum and maximum value
 - Percentiles
 - Variance and Standard Deviation
 - Covariance and Correlation
 - Probability distributions

The R language was developed by two statisticians. It has many built-in functionalities, in addition to libraries for the exact purpose of statistical analysis.

R Data Set

Data Set

A data set is a collection of data, often presented in a table.

There is a popular built-in data set in R called "**mtcars**" (Motor Trend Car Road Tests), which is retrieved from the 1974 Motor Trend US Magazine.

In the examples below (and for the next chapters), we will use the `mtcars` data set, for statistical purposes:

Example

```
# Print the mtcars data set  
mtcars
```

Result:

mpg cyl disp hp drat wt qsec vs am
gear carb

Car Model	MPG	Cylinders	Displacement (cu.in.)	Number of Doors	Weight (lb)	Acceleration (sec)	Transmission	Brake (ft)	MPG (Hwy)	MPG (City)	Wheelpitch
Mazda RX4 4 4	21.0	6	160.0	110	3.90	2.620	16.46	0	1		
Mazda RX4 Wag 4 4	21.0	6	160.0	110	3.90	2.875	17.02	0	1		
Datsun 710 4 1	22.8	4	108.0	93	3.85	2.320	18.61	1	1		
Hornet 4 Drive 3 1	21.4	6	258.0	110	3.08	3.215	19.44	1	0		
Hornet Sportabout 3 2	18.7	8	360.0	175	3.15	3.440	17.02	0	0		
Valiant 3 1	18.1	6	225.0	105	2.76	3.460	20.22	1	0		
Duster 360 3 4	14.3	8	360.0	245	3.21	3.570	15.84	0	0		
Merc 240D 4 2	24.4	4	146.7	62	3.69	3.190	20.00	1	0		
Merc 230 4 2	22.8	4	140.8	95	3.92	3.150	22.90	1	0		
Merc 280 4 4	19.2	6	167.6	123	3.92	3.440	18.30	1	0		
Merc 280C 4 4	17.8	6	167.6	123	3.92	3.440	18.90	1	0		
Merc 450SE 3 3	16.4	8	275.8	180	3.07	4.070	17.40	0	0		
Merc 450SL 3 3	17.3	8	275.8	180	3.07	3.730	17.60	0	0		
Merc 450SLC 3 3	15.2	8	275.8	180	3.07	3.780	18.00	0	0		
Cadillac Fleetwood 3 4	10.4	8	472.0	205	2.93	5.250	17.98	0	0		
Lincoln Continental 3 4	10.4	8	460.0	215	3.00	5.424	17.82	0	0		
Chrysler Imperial 3 4	14.7	8	440.0	230	3.23	5.345	17.42	0	0		
Fiat 128 4 1	32.4	4	78.7	66	4.08	2.200	19.47	1	1		
Honda Civic 4 2	30.4	4	75.7	52	4.93	1.615	18.52	1	1		
Toyota Corolla 4 1	33.9	4	71.1	65	4.22	1.835	19.90	1	1		
Toyota Corona 3 1	21.5	4	120.1	97	3.70	2.465	20.01	1	0		
Dodge Challenger 3 2	15.5	8	318.0	150	2.76	3.520	16.87	0	0		
AMC Javelin 3 2	15.2	8	304.0	150	3.15	3.435	17.30	0	0		
Camaro Z28 3 4	13.3	8	350.0	245	3.73	3.840	15.41	0	0		
Pontiac Firebird 3 2	19.2	8	400.0	175	3.08	3.845	17.05	0	0		
Fiat X1-9 4 1	27.3	4	79.0	66	4.08	1.935	18.90	1	1		

Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1
5 2									
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1
5 2									
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1
5 4									
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1
5 6									
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1
5 8									
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1

Information About the Data Set

You can use the question mark (?) to get information about the `mtcars` data set:

Example

```
# Use the question mark to get information about the data set
```

```
?mtcars
```

Result:

`mtcars` {datasets}

R Documentation

Motor Trend Car Road Tests

Description

The data was extracted from the 1974 *Motor Trend* US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973-74 models).

Usage

`mtcars`

Format

A data frame with 32 observations on 11 (numeric) variables.

[, 1] mpg Miles/(US) gallon

[, 2] cyl Number of cylinders

[, 3] disp Displacement (cu.in.)

[, 4] hp Gross horsepower

[, 5] drat Rear axle ratio

[, 6] wt Weight (1000 lbs)

[, 7] qsec 1/4 mile time

[, 8] vs Engine (0 = V-shaped, 1 = straight)

[, 9] am Transmission (0 = automatic, 1 = manual)

[,10] gear Number of forward gears

[,11] carb Number of carburetors

Note

Henderson and Velleman (1981) comment in a footnote to Table 1: 'Hocking [original transcriber]'s noncrucial coding of the Mazda's rotary engine as a straight six-cylinder engine and the Porsche's flat engine as a V engine, as well as the inclusion of the diesel Mercedes 240D, have been retained to enable direct comparisons to be made with previous analyses.'

Source

Henderson and Velleman (1981), Building multiple regression models interactively. *Biometrics*, **37**, 391-411.

Examples

```
require(graphics)
pairs(mtcars, main = "mtcars data", gap = 1/4)
coplot(mpg ~ disp | as.factor(cyl), data = mtcars,
       panel = panel.smooth, rows = 1)
## possibly more meaningful, e.g., for summary() or bivariate
plots:
mtcars2 <- within(mtcars, {
  vs <- factor(vs, labels = c("V", "S"))
  am <- factor(am, labels = c("automatic", "manual"))
  cyl <- ordered(cyl)
  gear <- ordered(gear)
  carb <- ordered(carb)}
```

```
})
summary(mtcars2)
```

Get Information

Use the `dim()` function to find the dimensions of the data set, and the `names()` function to view the names of the variables:

Example

```
Data_Cars <- mtcars # create a variable of the mtcars data set for
better organization

# Use dim() to find the dimension of the data set
dim(Data_Cars)

# Use names() to find the names of the variables from the data set
names(Data_Cars)
```

Result:

```
[1] 32 11
[1] "mpg"   "cyl"   "disp"  "hp"    "drat"  "wt"    "qsec" "vs"    "am"
"gear"
[11] "carb"
```

Use the `rownames()` function to get the name of each row in the first column, which is the name of each car:

Example

```
Data_Cars <- mtcars

rownames(Data_Cars)
```

Result:

```
[1] "Mazda RX4"           "Mazda RX4 Wag"      "Datsun 710"
[4] "Hornet 4 Drive"       "Hornet Sportabout" "Valiant"
[7] "Duster 360"          "Merc 240D"        "Merc 230"
[10] "Merc 280"            "Merc 280C"        "Merc 450SE"
[13] "Merc 450SL"          "Merc 450SLC"     "Cadillac
Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial" "Fiat 128"
[19] "Honda Civic"         "Toyota Corolla"   "Toyota Corona"
[22] "Dodge Challenger"    "AMC Javelin"     "Camaro Z28"
[25] "Pontiac Firebird"    "Fiat X1-9"       "Porsche 914-2"
[28] "Lotus Europa"        "Ford Pantera L"  "Ferrari Dino"
```

[31] "Maserati Bora"

"Volvo 142E"

From the examples above, we have found out that the data set has **32** observations (Mazda RX4, Mazda RX4 Wag, Datsun 710, etc) and **11** variables (mpg, cyl, disp, etc).

A variable is defined as something that can be measured or counted.

Here is a brief explanation of the variables from the mtcars data set:

Variable Name	Description
mpg	Miles/(US) Gallon
cyl	Number of cylinders
disp	Displacement
hp	Gross horsepower
drat	Rear axle ratio
wt	Weight (1000 lbs)
qsec	1/4 mile time
vs	Engine (0 = V-shaped, 1 = straight)

```
am           Transmission (0 = automatic, 1 = manual)
```

```
gear          Number of forward gears
```

```
carb          Number of carburetors
```

Print Variable Values

If you want to print all values that belong to a variable, access the data frame by using the `$` sign, and the name of the variable (for example `cyl` (cylinders)):

Example

```
Data_Cars <- mtcars
```

```
Data_Cars$cyl
```

Result:

```
[1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6  
8 4
```

Sort Variable Values

To sort the values, use the `sort()` function:

Example

```
Data_Cars <- mtcars
```

```
sort(Data_Cars$cyl)
```

Result:

```
[1] 4 4 4 4 4 4 4 4 4 4 4 6 6 6 6 6 6 6 8 8 8 8 8 8 8 8 8 8 8 8  
8 8
```

From the examples above, we see that most cars have 4 and 8 cylinders.

Analyzing the Data

Now that we have some information about the data set, we can start to analyze it with some statistical numbers.

For example, we can use the `summary()` function to get a statistical summary of the data:

Example

```
Data_Cars <- mtcars  
summary(Data_Cars)
```

Do not worry if you do not understand the output numbers. You will master them shortly.

The `summary()` function returns six statistical numbers for each variable:

- Min
- First quantile (percentile)
- Median
- Mean
- Third quantile (percentile)
- Max

We will cover all of them, along with other statistical numbers in the next chapters.

R Max and Min

Max Min

In the previous chapter, we introduced the **mtcars** data set. We will continue to use this data set throughout the next pages.

You learned from the [R Math](#) chapter that R has several built-in math functions. For example, the `min()` and `max()` functions can be used to find the lowest or highest value in a set:

Example

Find the largest and smallest value of the variable `hp` (horsepower).

```
Data_Cars <- mtcars
```

```
max(Data_Cars$hp)  
min(Data_Cars$hp)
```

Result:

```
[1] 335  
[1] 52
```

Now we know that the largest horsepower value in the set is **335**, and the lowest **52**.

We could take a look at the data set and try to find out which car these two values belongs to:

Observation of cars

Merc	450SLC		15.2	8	275.8	180	3.07	3.780	18.00	0	0
3	3										
Cadillac	Fleetwood		10.4	8	472.0	205	2.93	5.250	17.98	0	0
3	4										
Lincoln	Continental		10.4	8	460.0	215	3.00	5.424	17.82	0	0
3	4										
Chrysler	Imperial		14.7	8	440.0	230	3.23	5.345	17.42	0	0
3	4										
Fiat	128		32.4	4	78.7	66	4.08	2.200	19.47	1	1
4	1										
Honda	Civic		30.4	4	75.7	52	4.93	1.615	18.52	1	1
4	2										
Toyota	Corolla		33.9	4	71.1	65	4.22	1.835	19.90	1	1
4	1										
Toyota	Corona		21.5	4	120.1	97	3.70	2.465	20.01	1	0
3	1										
Dodge	Challenger		15.5	8	318.0	150	2.76	3.520	16.87	0	0
3	2										
AMC	Javelin		15.2	8	304.0	150	3.15	3.435	17.30	0	0
3	2										
Camaro	Z28		13.3	8	350.0	245	3.73	3.840	15.41	0	0
3	4										
Pontiac	Firebird		19.2	8	400.0	175	3.08	3.845	17.05	0	0
3	2										
Fiat	X1-9		27.3	4	79.0	66	4.08	1.935	18.90	1	1
4	1										
Porsche	914-2		26.0	4	120.3	91	4.43	2.140	16.70	0	1
5	2										
Lotus	Europa		30.4	4	95.1	113	3.77	1.513	16.90	1	1
5	2										
Ford	Pantera L		15.8	8	351.0	264	4.22	3.170	14.50	0	1
5	4										
Ferrari	Dino		19.7	6	145.0	175	3.62	2.770	15.50	0	1
5	6										
Maserati	Bora		15.0	8	301.0	335	3.54	3.570	14.60	0	1
5	8										
Volvo	142E		21.4	4	121.0	109	4.11	2.780	18.60	1	1
4	2										

By observing the table, it looks like the largest hp value belongs to a Maserati Bora, and the lowest belongs to a Honda Civic.

However, it is much easier (and safer) to let R find out this for us.

For example, we can use the `which.max()` and `which.min()` functions to find the index position of the max and min value in the table:

Example

```
Data_Cars <- mtcars
```

```
which.max(Data_Cars$hp)
which.min(Data_Cars$hp)
```

Result:

```
[1] 31
[1] 19
```

Or even better, combine `which.max()` and `which.min()` with the `rownames()` function to get the name of the car with the largest and smallest horsepower:

Example

```
Data_Cars <- mtcars
```

```
rownames(Data_Cars)[which.max(Data_Cars$hp)]
rownames(Data_Cars)[which.min(Data_Cars$hp)]
```

Result:

```
[1] "Maserati Bora"
[1] "Honda Civic"
```

Now we know for sure:

Maserati Bora is the car with the highest horsepower, and **Honda Civic** is the car with the lowest horsepower.

Outliers

Max and min can also be used to detect **outliers**. An outlier is a data point that differs from rest of the observations.

Example of data points that could have been outliers in the **mtcars** data set:

- If maximum of forward gears of a car was 11
- If minimum of horsepower of a car was 0
- If maximum weight of a car was 50 000 lbs

R Mean

Mean, Median, and Mode

In statistics, there are often three values that interests us:

- **Mean** - The average value
- **Median** - The middle value
- **Mode** - The most common value

Mean

To calculate the average value (mean) of a variable from the `mtcars` data set, find the sum of all values, and divide the sum by the number of values.

Sorted observation of wt (weight)

1.513	1.615	1.835	1.935	2.140	2.200	2.320	2.465
2.620	2.770	2.780	2.875	3.150	3.170	3.190	3.215
3.435	3.440	3.440	3.440	3.460	3.520	3.570	3.570
3.730	3.780	3.840	3.845	4.070	5.250	5.345	5.424

Luckily for us, the `mean()` function in R can do it for you:

Example

Find the average weight (`wt`) of a car:

```
Data_Cars <- mtcars
```

```
mean(Data_Cars$wt)
```

Result:

```
[1] 3.21725
```

R Median

Median

The median value is the value in the middle, after you have sorted all the values.

If we take a look at the values of the `wt` variable (from the `mtcars` data set), we will see that there are two numbers in the middle:

Sorted observation of wt (weight)

1.513	1.615	1.835	1.935	2.140	2.200	2.320	2.465
2.620	2.770	2.780	2.875	3.150	3.170	3.190	3.215
3.435	3.440	3.440	3.440	3.460	3.520	3.570	3.570
3.730	3.780	3.840	3.845	4.070	5.250	5.345	5.424

Note: If there are two numbers in the middle, you must divide the sum of those numbers by two, to find the median.

Luckily, R has a function that does all of that for you: Just use the `median()` function to find the middle value:

Example

Find the mid point value of weight (`wt`):

```
Data_Cars <- mtcars
```

```
median(Data_Cars$wt)
```

Result:

```
[1] 3.325
```

R Mode

Mode

The mode value is the value that appears the most number of times.

R does not have a function to calculate the mode. However, we can create our own function to find it.

If we take a look at the values of the `wt` variable (from the `mtcars` data set), we will see that the numbers **3.440** are often shown:

Sorted observation of wt (weight)

1.513	1.615	1.835	1.935	2.140	2.200	2.320	2.465
2.620	2.770	2.780	2.875	3.150	3.170	3.190	3.215
3.435	3.440	3.440	3.440	3.460	3.520	3.570	3.570
3.730	3.780	3.840	3.845	4.070	5.250	5.345	5.424

Instead of counting it ourselves, we can use the following code to find the mode:

Example

```
Data_Cars <- mtcars  
names(sort(-table(Data_Cars$wt)))[1]
```

Result:

```
[1] "3.44"
```

From the example above, we now know that the number that appears the most number of times in mtcars wt variable is **3.44** or **3.440 lbs.**

R Percentiles

Percentiles

$$P = (n/N) \times 100$$

What is the percentile value for the value 60 in a given population of weights of persons 50, 55, 40, 60, 100, 95, 90, 60, 80, 75.

Solution:

The given data is not sorted. So first sort the data in ascending order.

Sorted data: 40,50,55,60,60,75,80,90,95,100

Number of values fall under 60 (n)= 3

Total count of values (N)= 10

$$\text{Percentile} = (n/N) \times 100$$

$$= (3/10) \times 100$$

$$= (.3) \times 100$$

$$= 30$$

Percentiles are used in statistics to give you a number that describes the value that a given percent of the values are lower than.

If we take a look at the values of the `wt` (weight) variable from the `mtcars` data set:

Observation of wt (weight)

1.513	1.615	1.835	1.935	2.140	2.200	2.320	2.465
2.620	2.770	2.780	2.875	3.150	3.170	3.190	3.215
3.435	3.440	3.440	3.440	3.460	3.520	3.570	3.570
3.730	3.780	3.840	3.845	4.070	5.250	5.345	5.424

Example

```
Data_Cars <- mtcars
```

```
# c() specifies which percentile you want  
quantile(Data_Cars$wt, c(0.75))
```

Result:

```
75%  
3.61
```

If you run the `quantile()` function without specifying the `c()` parameter, you will get the percentiles of 0, 25, 50, 75 and 100:

Example

```
Data_Cars <- mtcars
```

```
quantile(Data_Cars$wt)
```

Result:

```
0%      25%      50%      75%      100%  
1.51300 2.58125 3.32500 3.61000 5.42400
```

Quartiles

Quartiles are data divided into four parts, when sorted in an ascending order:

1. The value of the first quartile cuts off the first 25% of the data
2. The value of the second quartile cuts off the first 50% of the data
3. The value of the third quartile cuts off the first 75% of the data
4. The value of the fourth quartile cuts off the 100% of the data

Use the `quantile()` function to get the quartiles.

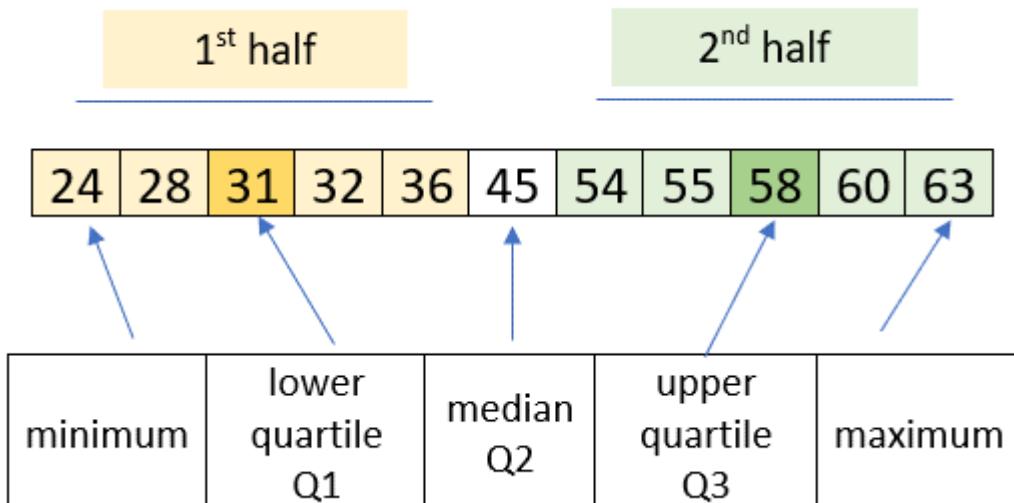


Quartile Formula

$$\text{Lower Quartile (Q1)} = (N+1) \times \frac{1}{4}$$

$$\text{Middle Quartile (Q2)} = (N+1) \times \frac{2}{4}$$

$$\text{Upper Quartile (Q3)} = (N+1) \times \frac{3}{4}$$



Quartile

There are several **quartiles** of an observation variable. The **first quartile**, or **lower quartile**, is the value that cuts off the first 25% of the data when it is sorted in ascending order. The **second quartile**, or **median**, is the value that cuts off the first 50%. The **third quartile**, or **upper quartile**, is the value that cuts off the first 75%.

Problem

Find the quartiles of the eruption durations in the data set `faithful`.

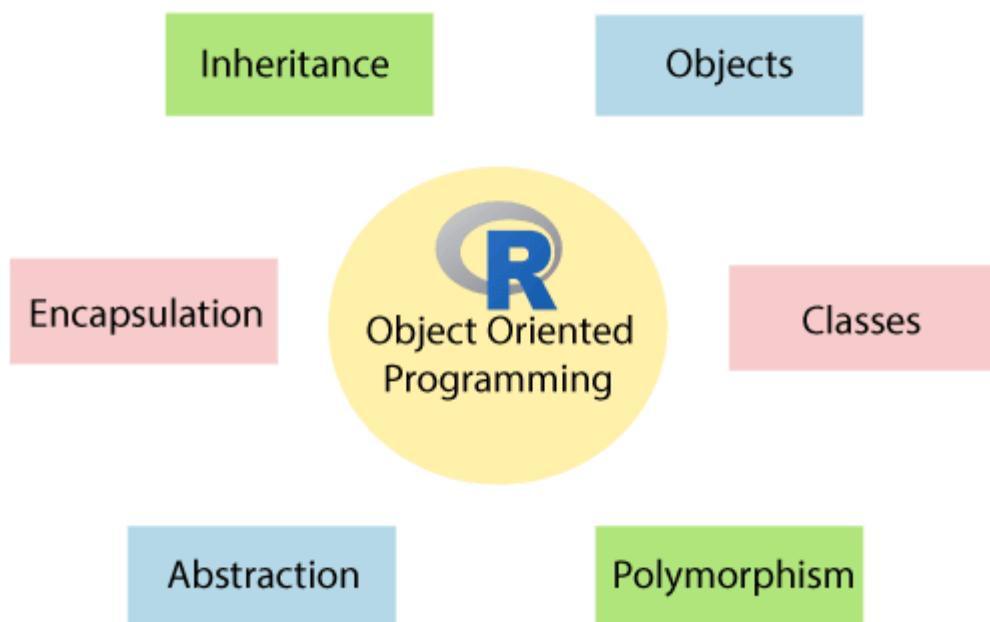
Solution

We apply the `quantile` function to compute the quartiles of eruptions.

```
> duration = faithful$eruptions      # the eruption durations  
> quantile(duration)             # apply the quantile function  
  0%    25%    50%    75%   100%  
1.6000 2.1627 4.0000 4.4543 5.1000
```

What is Object-Oriented Programming in R?

Object-Oriented Programming (OOP) is the most popular programming language. With the help of oops concepts, we can construct the modular pieces of code which are used to build blocks for large systems. R is a functional language, and we can do programming in oops style. In R, oops is a great tool to manage the complexity of larger programs.



In Object-Oriented Programming, S3 and S4 are the two important systems.

S3

In oops, the S3 is used to overload any function. So that we can call the functions with different names and it depends on the type of input parameter or the number of parameters.

S4

S4 is the most important characteristic of oops. However, this is a limitation, as it is quite difficult to debug. There is an optional reference class for S4.

Objects and Classes in R

In R, everything is an object. Therefore, programmers perform OOPS concept when they write code in R. An object is a data structure which has some methods that can act upon its attributes.

In R, classes are the outline or design for the object. Classes encapsulate the data members, along with the functions. In R, there are two most important classes, i.e., S3 and S4, which play an important role in performing OOPs concepts.

Let's discuss both the classes one by one with their examples for better understanding.

1) S3 Class

With the help of the S3 class, we can take advantage of the ability to implement the generic function OO. Furthermore, using only the first argument, S3 is capable of dispatching. S3 differs from traditional programming languages such as Java, C ++, and C #, which implement OO passing messages. This makes S3 easy to implement. In the S3 class, the generic function calls the method. S3 is very casual and has no formal definition of classes.

S3 requires very little knowledge from the programmer.

Creating an S3 class

In R, we define a function which will create a class and return the object of the created class. A list is made with relevant members, class of the list is determined, and a copy of the list is returned. There is the following syntax to create a class

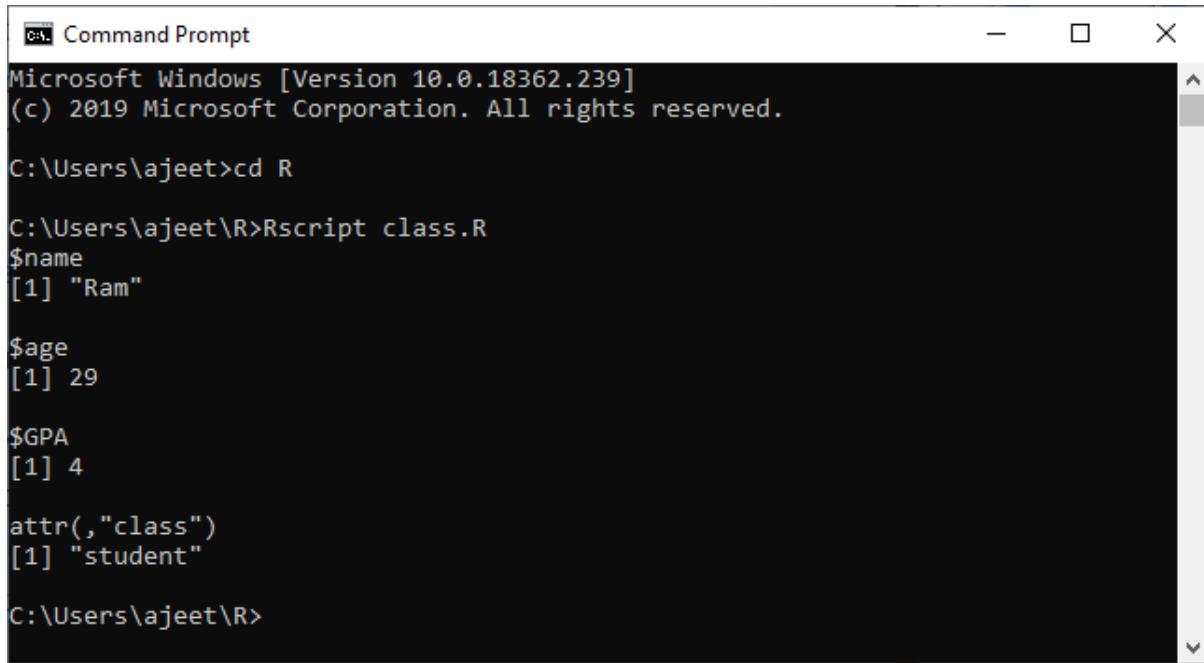
1. variable_name <- list(member1, member2, member3.....memberN)

Example

```
s <- list(name = "Ram", age = 29, GPA = 4.0)
class(s) <- "Faculty"
```

S

Output



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window displays the following text:

```
Microsoft Windows [Version 10.0.18362.239]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\ajeet>cd R

C:\Users\ajeet\R>Rscript class.R
$name
[1] "Ram"

$age
[1] 29

$GPA
[1] 4

attr(,"class")
[1] "student"

C:\Users\ajeet\R>
```

There is the following way in which we define our generic function print.

1. print
2. function(x, ...)
3. UseMethod("Print")

When we execute or run the above code, it will give us the following output:

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
[Previously saved workspace restored]
> print
function (x, ...)
UseMethod("print")
<bytecode: 0x0000000014d3c578>
<environment: namespace:base>
> function (x, ...)
+ UseMethod("print")
function (x, ...)
UseMethod("print")
> -

Like print function, we will make a generic function GPA to assign a new value to our GPA member. In the following way we will make the generic function GPA

1. GPA <- function(obj1){
2. UseMethod("GPA")
3. }

Once our generic function GPA is created, we will implement a default function for it

1. GPA.default <- function(obj){
2. cat("We are entering in generic function\n")
3. }

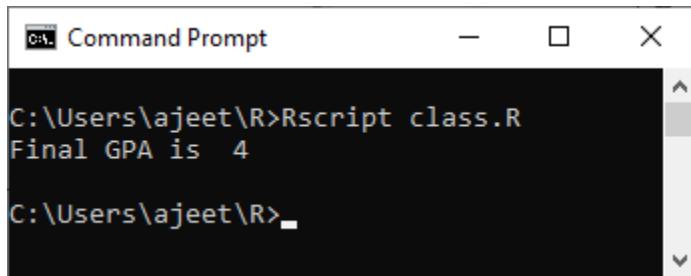
After that we will make a new method for our GPA function in the following way

1. GPA.faculty <- function(obj1){
2. cat("Final GPA is ",obj1\$GPA,"\n")
3. }

And at last we will run the method GPA as

1. GPA(s)

Output



```
C:\Users\ajeet\R>Rscript class.R
Final GPA is 4
C:\Users\ajeet\R>
```

Inheritance in S3

Inheritance means extracting the features of one class into another class. In the S3 class of R, inheritance is achieved by applying the class attribute in a vector.

For inheritance, we first create a function which creates new object of class faculty in the following way

```
faculty<- function(n,a,g) {
  value <- list(nname=n, aage=a, GPA=g)
  attr(value, "class") <- "faculty"
  value
}
```

After that we will define a method for generic function print() as

```
print.student <- function(obj1) {
  cat(1obj$name, "\n")
  cat(1obj$age, "years old\n")
  cat("GPA:", obj1$GPA, "\n")
}
```

Now, we will create an object of class InternationalFaculty which will inherit from faculty class. This process will be done by assigning a character vector of class name as:

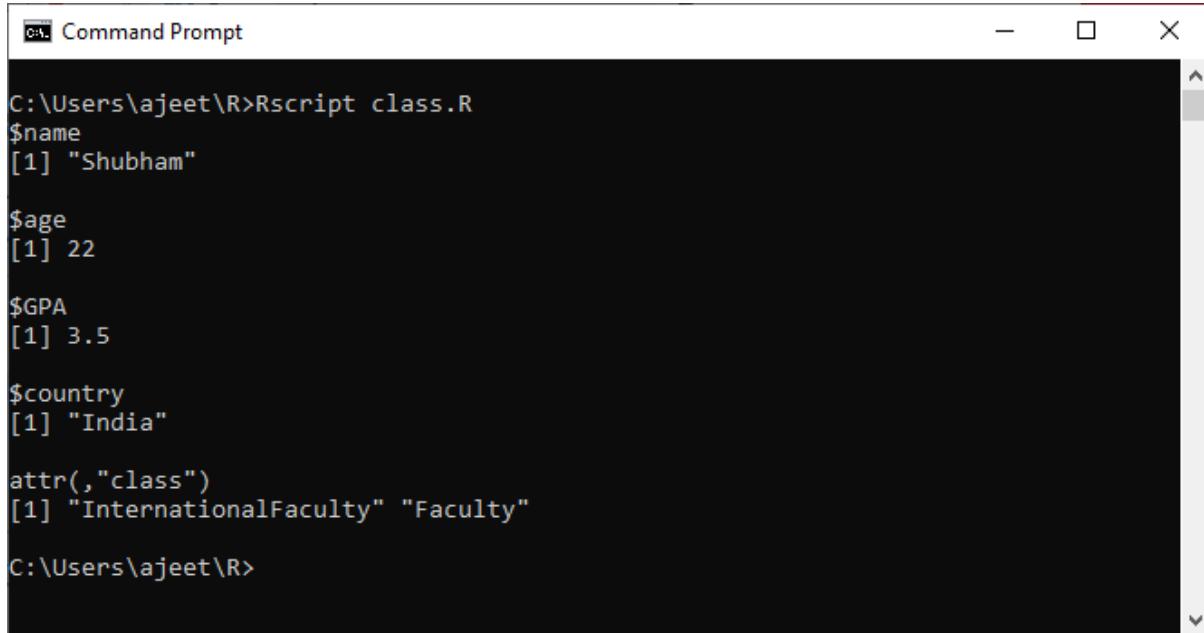
1. class(Objet) <- c(child, parent)

so,

1. # create a list
2. fac <- list(**name**="Shubham", **age**=22, **GPA**=3.5, **country**="India")
3. # make it of the class InternationalFaculty which is derived from the class Faculty

4. class(fac) <- c("InternationalFaculty", "Faculty")
5. # print it out
6. fac

When we run the above code which we have discussed, it will generate the following output:



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window contains the following text output from an R script:

```
C:\Users\ajeet\R>Rscript class.R
$name
[1] "Shubham"

$age
[1] 22

$GPA
[1] 3.5

$country
[1] "India"

attr(,"class")
[1] "InternationalFaculty" "Faculty"
C:\Users\ajeet\R>
```

We can see above that, we have not defined any method of form print.InternationalFaculty(), the method called print.Faculty(). This method of class Faculty was inherited.

So our next step is to defined print.InternationalFaculty() in the following way:

1. print.InternationalFaculty<- function(obj1) {
2. cat(obj1\$name, "is from", obj1\$country, "\n")
3. }

The above function will overwrite the method defined for class faculty as

1. Fac

```
C:\ Command Prompt
C:\Users\ajeet\R>Rscript class.R
$name
[1] "Shubham"

$age
[1] 22

$GPA
[1] 3.5

$country
[1] "India"

attr(,"class")
[1] "InternationalFaculty" "Faculty"
Shubham is from India

C:\Users\ajeet\R>
```

getS3method and getAnywhere function

There are the two most common and popular S3 method functions which are used in R. The first method is **getS3method()** and the second one is **getAnywhere()**.

S3 finds the appropriate method associated with a class, and it is useful to see how a method is implemented. Sometimes, the methods are non-visible, because they are hidden in a namespace. We use `getS3method` or `getAnywhere` to solve this problem.

getS3method

```

Rterm (64-bit)
[1] FALSE
> getS3method("predict","ppr")
function (object, newdata, ...)
{
  if (missing(newdata))
    return(fitted(object))
  if (!is.null(object$terms)) {
    newdata <- as.data.frame(newdata)
    rn <- row.names(newdata)
    Terms <- delete.response(object$terms)
    m <- model.frame(Terms, newdata, na.action = na.omit,
      xlev = object$xlevels)
    if (!is.null(cl <- attr(Terms, "dataClasses")))
      .checkMFClasses(cl, m)
    keep <- match(row.names(m), rn)
    x <- model.matrix(Terms, m, contrasts.arg = object$contrasts)
  }
  else {
    x <- as.matrix(newdata)
    keep <- seq_len(nrow(x))
    rn <- dimnames(x)[[1L]]
  }
  if (ncol(x) != object$p)
    stop("wrong number of columns in 'x'")
  res <- matrix(NA, length(keep), object$q, dimnames = list(rn,
    object$ynames))
  res[keep, ] <- matrix(.Fortran(C_pppred, as.integer(nrow(x)),
    as.double(x), as.double(object$smod), y = double(nrow(x)) *
      object$q), double(2 * object$smod[4L]))$y, ncol = object$q)
  drop(res)
}
<bytecode: 0x0000000004ae3608>
<environment: namespace:stats>
> -

```

getAnywhere function

1. getAnywhere("simpleloess")

2) S4 Class

The S4 class is similar to the S3 but is more formal than the latter one. It differs from S3 in two different ways. First, in S4, there are formal class definitions which provide a description and representation of classes. In addition, it has special auxiliary functions for defining methods and generics. The S4 also offers multiple dispatches. This means that common functions are capable of taking methods based on multiple arguments which are based on class.

Creating an S4 class

In R, we use `setClass()` command for creating S4 class. In S4 class, we will specify a function for verifying the data consistency and also specify the default value. In R, member variables are called slots.

To create an S3 class, we have to define the class and its slots. There are the following steps to create an S4 class

Step 1:

In the first step, we will create a new class called faculty with three slots name, age, and GPA.

1. `setClass("faculty", slots=list(name="character", age="numeric", GPA="numeric"))`

There are many other optional arguments of `setClass()` function which we can explore by using `?setClass` command.

R: Create a Class Definition

127.0.0.1:17387/library/methods/html/setClass.html

setClass {methods}

Create a Class Definition

Description

Create a class definition and return a generator function to create objects from the class. Typical usage will be of the style:

```
myClass <- setClass("myClass", slots= ...., contains =....)
```

where the first argument is the name of the new class and, if supplied, the arguments slots= and contains= specify the slots in the new class and existing classes from which the new class should inherit. Calls to `setClass()` are normally found in the source of a package; when the package is loaded the class will be defined in the package's namespace. Assigning the generator function with the name of the class is convenient for users, but not a requirement.

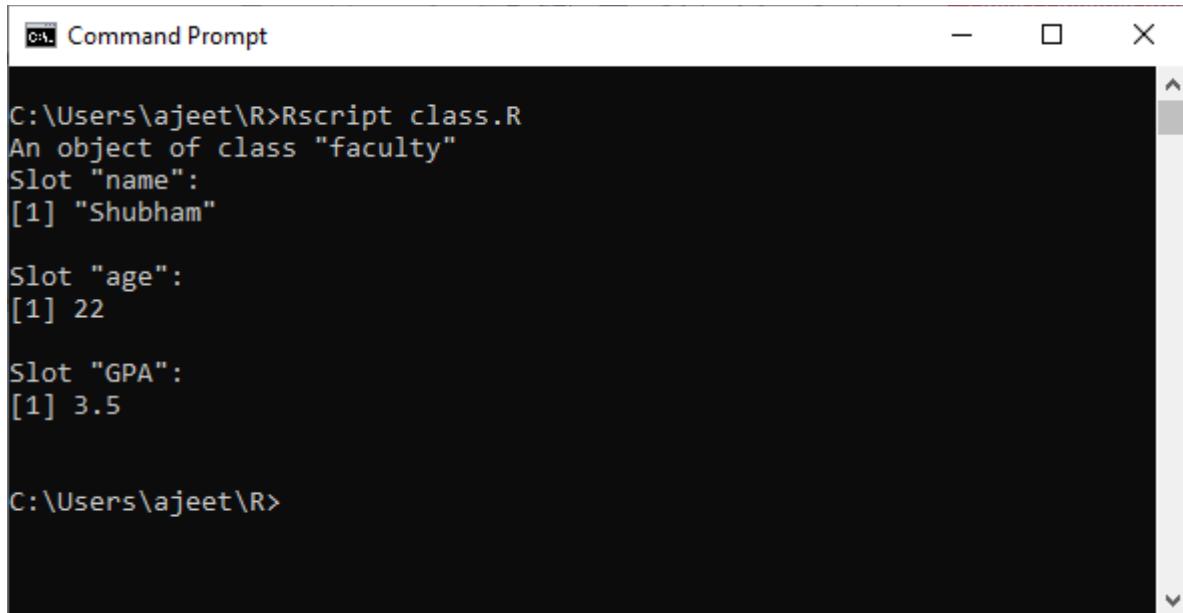
Usage

Step 2:

In the next step, we will create the object of S4 class. R provides `new()` function to create an object of S4 class. In this `new` function we pass the class name and the values for the slots in the following way:

1. `setClass("faculty", slots=list(name="character", age="numeric", GPA="numeric"))`
2. # creating an object using `new()`
3. # providing the class name and value for slots
4. `s <- new("faculty", name="Shubham", age=22, GPA=3.5)`
5. `s`

It will generate the following output



```
C:\Users\ajeet\R>Rscript class.R
An object of class "faculty"
Slot "name":
[1] "Shubham"

Slot "age":
[1] 22

Slot "GPA":
[1] 3.5

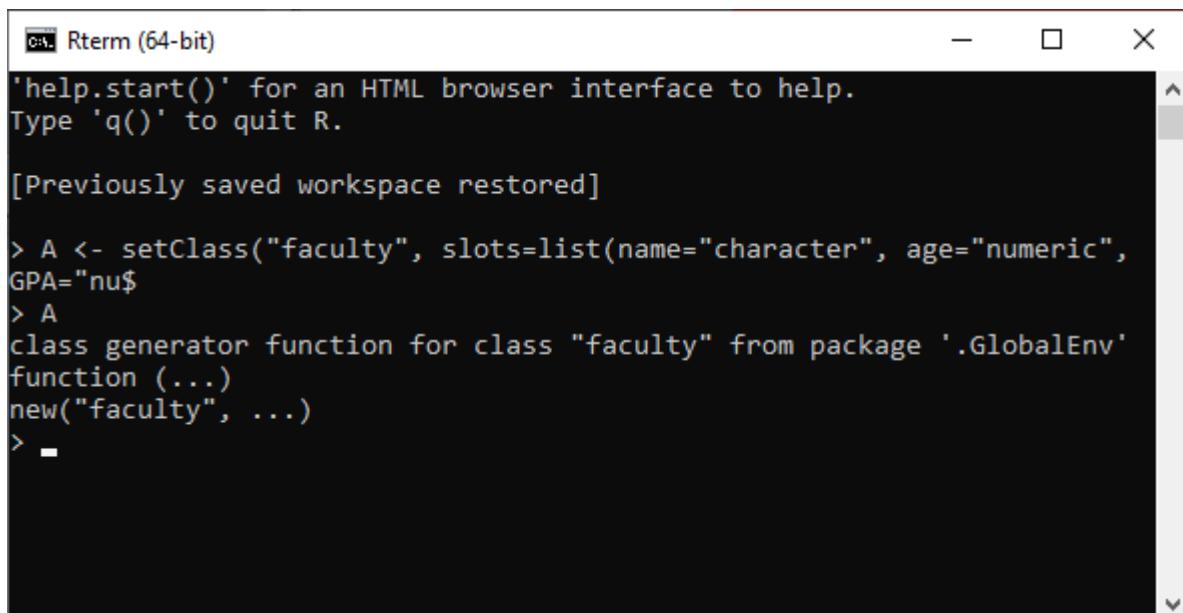
C:\Users\ajeet\R>
```

Creating S4 objects using a generator function

The `setClass()` function returns a generator function. This generator function helps in creating new objects. And it acts as a constructor.

1. A `<- setClass("faculty", slots=list(name="character", age="numeric", GPA="numeric"))`
2. A

It will generate the following output:



```
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

> A <- setClass("faculty", slots=list(name="character", age="numeric",
GPA="nu$)
> A
class generator function for class "faculty" from package '.GlobalEnv'
function (...)

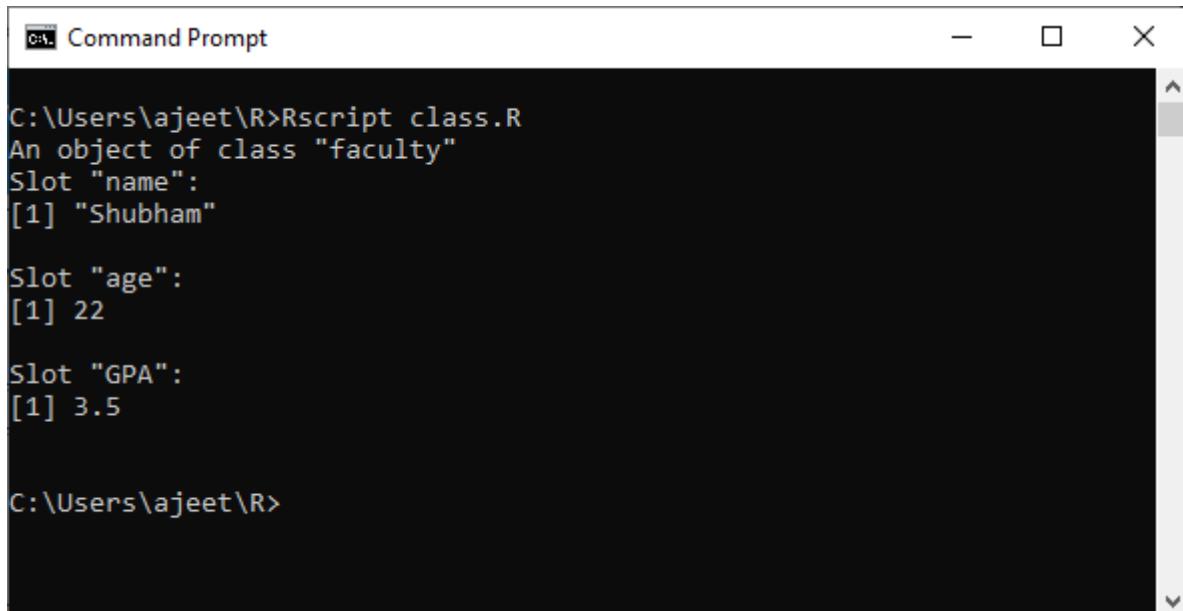
new("faculty", ...)
> -
```

Now we can use the above constructor function to create new objects. The constructor in turn uses the new() function to create objects. It is just a wrap around. Let's see an example to understand how S4 object is created with the help of generator function.

Example

1. faculty<-
`setClass("faculty", slots=list(name="character", age="numeric", GPA="numeric"))`
2. # creating an object using generator() function
3. # providing the class name and value for slots
4. faculty(name="Shubham", age=22, GPA=3.5)

Output



The screenshot shows an R command prompt window titled "Command Prompt". The console output is as follows:

```
C:\Users\ajeet\R>Rscript class.R
An object of class "faculty"
Slot "name":
[1] "Shubham"

Slot "age":
[1] 22

Slot "GPA":
[1] 3.5

C:\Users\ajeet\R>
```

Inheritance in S4 class

Like S3 class, we can perform inheritance in S4 class also. The derived class will inherit both attributes and methods of the parent class. Let's start understanding that how we can perform inheritance in S4 class. There are the following ways to perform inheritance in S4 class:

Step 1:

In the first step, we will create or define class with appropriate slots in the following way:

1. `setClass("faculty",`

2. `slots=list(name="character", age="numeric", GPA="numeric")`
3. `)`

Step 2:

After defining class, our next step is to define class method for the `display()` generic function. This will be done in the following manner:

1. `setMethod("show",`
2. `"faculty",`
3. `function(obj) {`
4. `cat(obj@name, "\n")`
5. `cat(obj@age, "years old\n")`
6. `cat("GPA:", obj@GPA, "\n")`
7. `})`

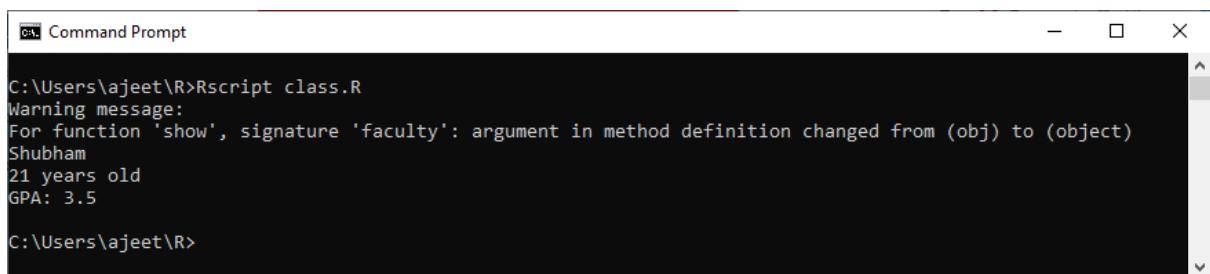
Step 3:

In the next step, we will define the derived class with the argument `contains`. The derived class is defined in the following way

1. `setClass("Internationalfaculty",`
2. `slots=list(country="character"),`
3. `contains="faculty"`
4. `)`

In our derived class we have defined only one attribute i.e. `country`. Other attributes will be inherited from its parent class.

```
s <- new("Internationalfaculty", name="John", age=21, GPA=3.5, country="India")
show(s)
```



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command `Rscript class.R` is run, resulting in the following output:

```
C:\Users\ajeet\R>Rscript class.R
Warning message:
For function 'show', signature 'faculty': argument in method definition changed from (obj) to (object)
Shubham
21 years old
GPA: 3.5
C:\Users\ajeet\R>
```

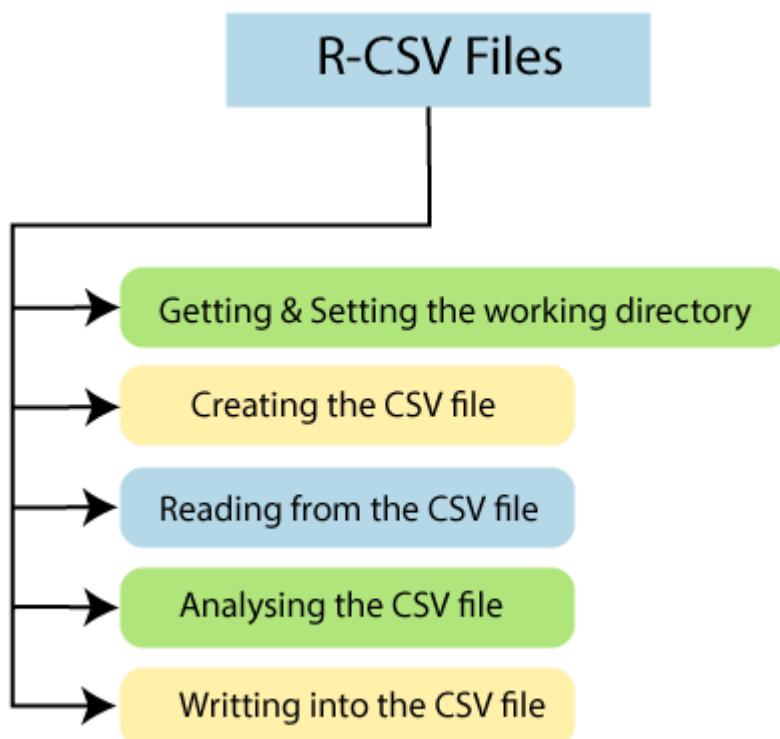
R CSV Files

A **Comma-Separated Values (CSV) file** is a plain text file which contains a list of data. These files are often used for the exchange of data between different applications. For example, databases and contact managers mostly support CSV files.

These files can sometimes be called **character-separated values** or **comma-delimited files**. They often use the comma character to separate data, but sometimes use other characters such as semicolons. The idea is that we can export the complex data from one application to a CSV file, and then importing the data in that CSV file to another application.

Storing data in excel spreadsheets is the most common way for data storing, which is used by the data scientists. There are lots of packages in R designed for accessing data from the excel spreadsheet. Users often find it easier to save their spreadsheets in comma-separated value files and then use R's built-in functionality to read and manipulate the data.

R allows us to read data from files which are stored outside the R environment. Let's start understanding how we can read and write data into CSV files. The file should be present in the current working directory so that R can read it. We can also set our directory and read file from there.



Getting and setting the working directory

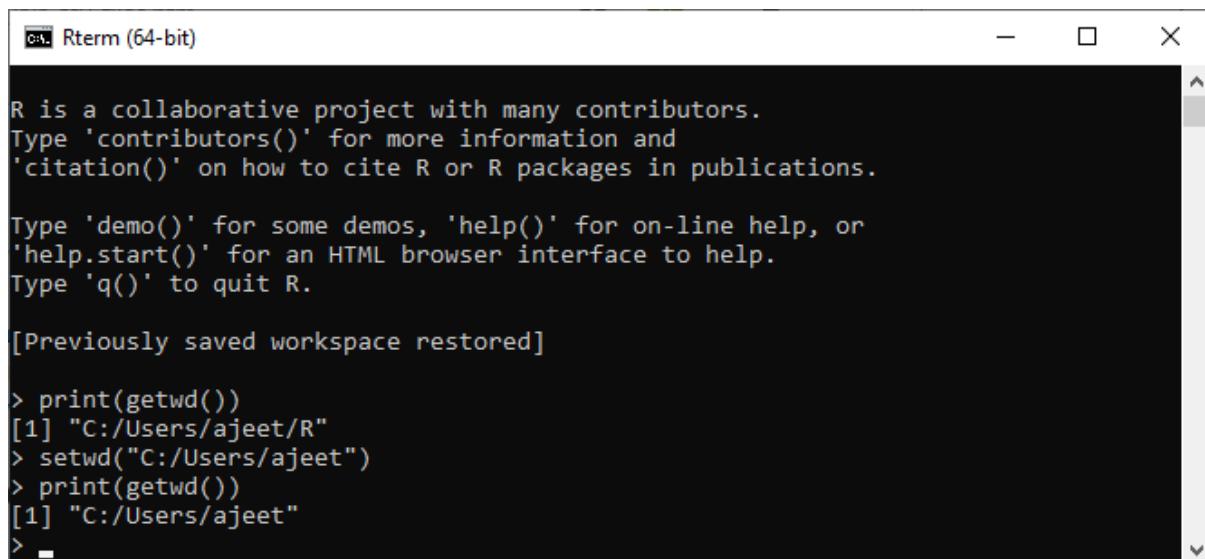
In R, getwd() and setwd() are the two useful functions. The getwd() function is used to check on which directory the R workspace is pointing. And the setwd() function is used to set a new working directory to read and write files from that directory.

Let's see an example to understand how getwd() and setwd() functions are used.

Example

1. # Getting and printing current working directory.
2. print(getwd())
3. # Setting the current working directory.
4. setwd("C:/Users/ajeet")
5. # Getting and printing the current working directory.
6. print(getwd())

Output



R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

> print(getwd())
[1] "C:/Users/ajeet/R"
> setwd("C:/Users/ajeet")
> print(getwd())
[1] "C:/Users/ajeet"
>

Creating a CSV File

A text file in which a comma separates the value in a column is known as a CSV file. Let's start by creating a CSV file with the help of the data, which is mentioned below by saving with .csv extension using the save As All files (*.*) option in the notepad.

Example: record.csv

1. id,name,salary,start_date,dept
2. 1,Shubham,613.3,2012-01-01,IT
3. 2,Arpita,525.2,2013-09-23,Operations

4. 3,Vaishali,63,2014-11-15,IT
5. 4,Nishka,749,2014-05-11,HR
6. 5,Gunjan,863.25,2015-03-27,Finance
7. 6,Sumit,588,2013-05-21,IT
8. 7,Anisha,932.8,2013-07-30,Operations
9. 8,Akash,712.5,2014-06-17,Financ

Output

```

C:\Users\ajeet\R\record.csv - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
record.csv x
1 id,name,salary,start_date,dept
2 1,Shubham,613.3,2012-01-01,IT
3 2,Arpita,525.2,2013-09-23,Operations
4 3,Vaishali,63,2014-11-15,IT
5 4,Nishka,749,2014-05-11,HR
6 5,Gunjan,863.25,2015-03-27,Finance
7 6,Sumit,588,2013-05-21,IT
8 7,Anisha,932.8,2013-07-30,Operations
9 8,Akash,712.5,2014-06-17,Financ
10
length : 292 lines Ln : 10 Col : 1 Sel : 0 | 0 Windows (CR LF) UTF-8 INS ...

```

Reading a CSV file

R has a rich set of functions. R provides `read.csv()` function, which allows us to read a CSV file available in our current working directory. This function takes the file name as an input and returns all the records present on it.

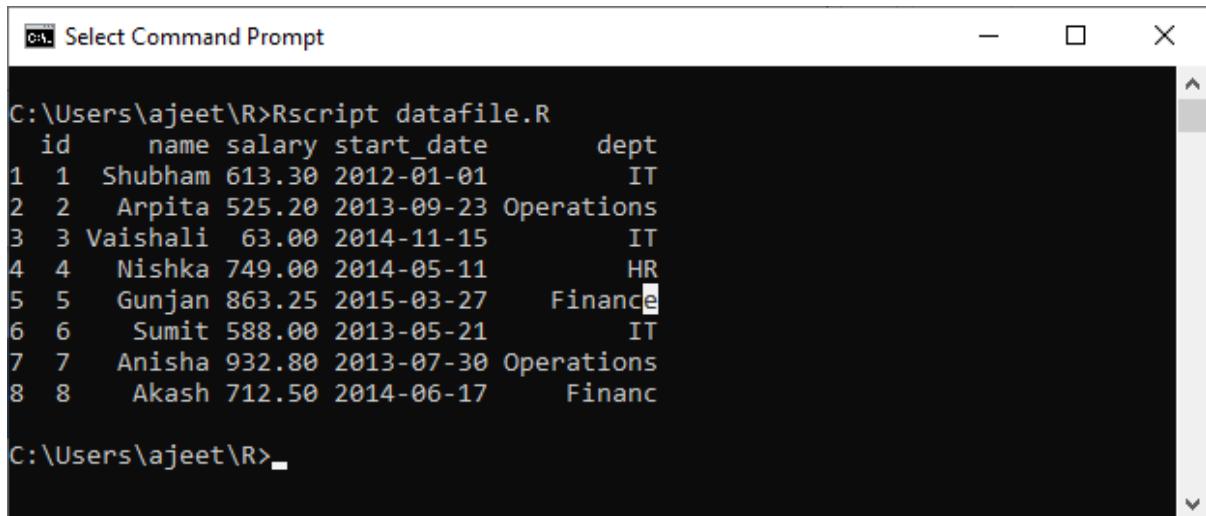
Let's use our `record.csv` file to read records from it using `read.csv()` function.

Example

1. `data <- read.csv("record.csv")`
2. `print(data)`

When we execute above code, it will give the following output

Output



```
C:\Users\ajeet\R>Rscript datafile.R
  id      name salary start_date      dept
1  1 Shubham 613.30 2012-01-01        IT
2  2 Arpita 525.20 2013-09-23 Operations
3  3 Vaishali 63.00 2014-11-15        IT
4  4 Nishka 749.00 2014-05-11       HR
5  5 Gunjan 863.25 2015-03-27 Finance
6  6 Sumit 588.00 2013-05-21        IT
7  7 Anisha 932.80 2013-07-30 Operations
8  8 Akash 712.50 2014-06-17 Financ
C:\Users\ajeet\R>
```

Analyzing the CSV File

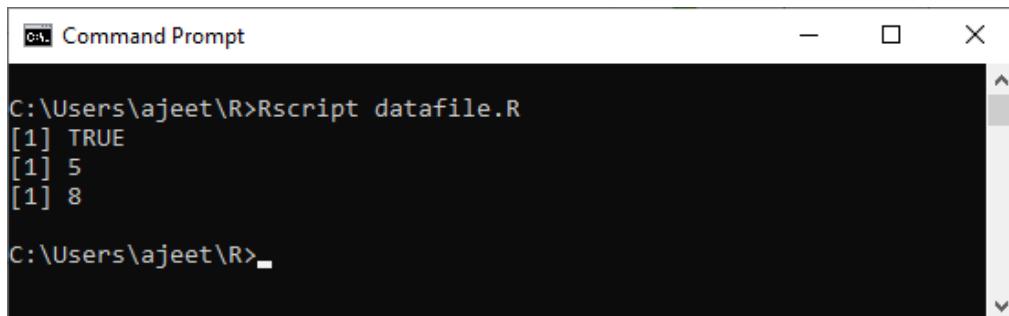
When we read data from the .csv file using **read.csv()** function, by default, it gives the output as a data frame. Before analyzing data, let's start checking the form of our output with the help of **is.data.frame()** function. After that, we will check the number of rows and number of columns with the help of **nrow()** and **ncol()** function.

Example

1. csv_data<- read.csv("record.csv")
2. print(is.data.frame(csv_data))
3. print(ncol(csv_data))
4. print(nrow(csv_data))

When we run above code, it will generate the following output:

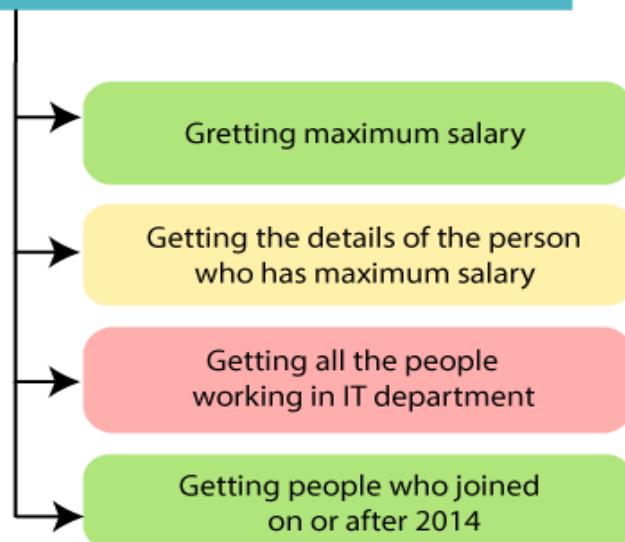
Output



```
C:\Users\ajeet\R>Rscript datafile.R
[1] TRUE
[1] 5
[1] 8
C:\Users\ajeet\R>
```

From the above output, it is clear that our data is read in the form of the data frame. So we can apply all the functions of the data frame, which we have discussed in the earlier sections.

Analysing the CSV file



Example: Getting the maximum salary

1. # Creating a data frame.
2. csv_data<- read.csv("record.csv")
- 3.
4. # Getting the maximum salary from data frame.
5. max_sal<- max(csv_data\$salary)
6. print(max_sal)

Output

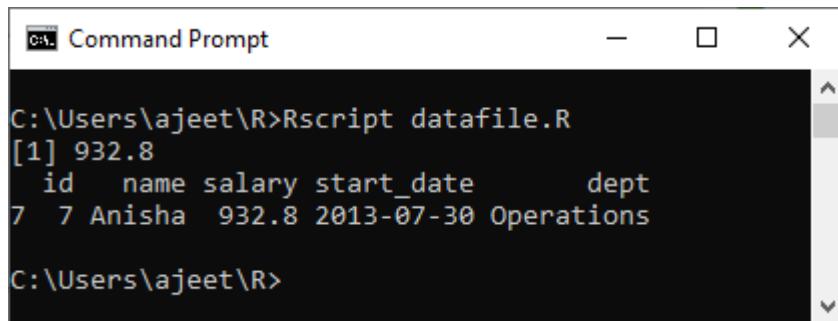
```
Command Prompt
C:\Users\ajeet\R>Rscript datafile.R
[1] 932.8
C:\Users\ajeet\R>
```

Example: Getting the details of the person who have a maximum salary

1. # Creating a data frame.
2. csv_data<- read.csv("record.csv")
- 3.
4. # Getting the maximum salary from data frame.
5. max_sal<- max(csv_data\$salary)

```
6. print(max_sal)
7.
8. #Getting the details of the person who have maximum salary
9. details <- subset(csv_data,salary==max(salary))
10. print(details)
```

Output



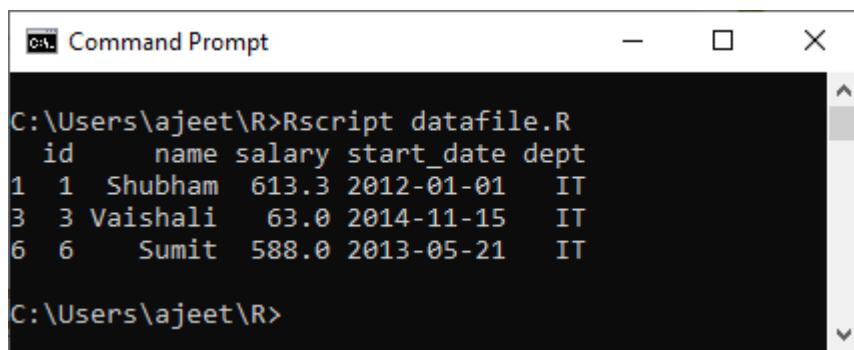
A screenshot of a Windows Command Prompt window titled "Command Prompt". The window shows the following text output:

```
C:\Users\ajeet\R>Rscript myfile.R
[1] 932.8
  id   name salary start_date      dept
7  7 Anisha  932.8 2013-07-30 Operations
C:\Users\ajeet\R>
```

Example: Getting the details of all the persons who are working in the IT department

```
1. # Creating a data frame.
2. csv_data<- read.csv("record.csv")
3.
4. #Getting the details of all the persons who are working in IT department
5. details <- subset(csv_data,dept=="IT")
6. print(details)
```

Output



A screenshot of a Windows Command Prompt window titled "Command Prompt". The window shows the following text output:

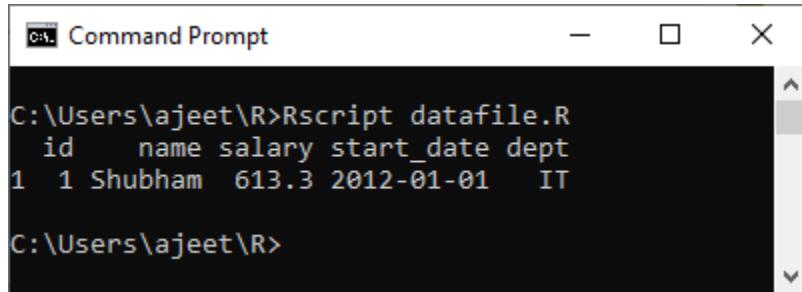
```
C:\Users\ajeet\R>Rscript myfile.R
  id   name salary start_date dept
1  1 Shubham  613.3 2012-01-01  IT
3  3 Vaishali  63.0 2014-11-15  IT
6  6 Sumit   588.0 2013-05-21  IT
C:\Users\ajeet\R>
```

Example: Getting the details of the persons whose salary is greater than 600 and working in the IT department.

```
1. # Creating a data frame.
2. csv_data<- read.csv("record.csv")
```

- 3.
4. #Getting the details of all the people who are working in IT department
5. details <- subset(csv_data,dept=="IT"&salary>600)
6. print(details)

Output

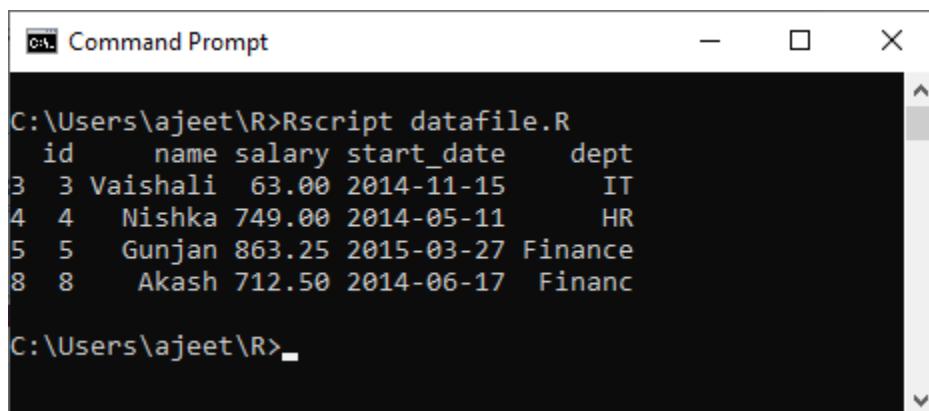


```
Command Prompt
C:\Users\ajeet\R>Rscript myfile.R
  id      name salary start_date dept
1  1 Shubham   613.3 2012-01-01    IT
C:\Users\ajeet\R>
```

Example: Getting details of those peoples who joined on or after 2014.

1. # Creating a data frame.
2. csv_data<- read.csv("record.csv")
- 3.
4. #Getting details of those peoples who joined on or after 2014
5. details <- subset(csv_data,as.Date(start_date)>as.Date("2014-01-01"))
6. print(details)

Output



```
Command Prompt
C:\Users\ajeet\R>Rscript myfile.R
  id      name salary start_date     dept
3  3 Vaishali   63.00 2014-11-15      IT
4  4 Nishka  749.00 2014-05-11      HR
5  5 Gunjan  863.25 2015-03-27 Finance
8  8 Akash  712.50 2014-06-17 Financ
C:\Users\ajeet\R>
```

Writing into a CSV file

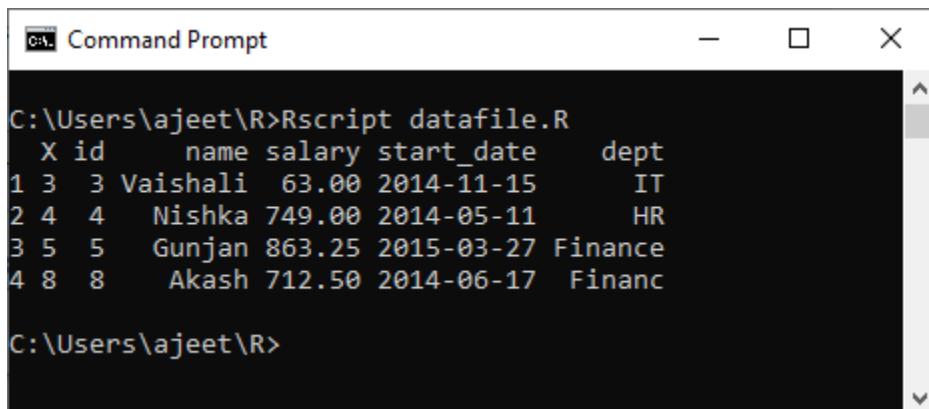
Like reading and analyzing, R also allows us to write into the .csv file. For this purpose, R provides a `write.csv()` function. This function creates a CSV file from an existing data frame. This function creates the file in the current working directory.

Let's see an example to understand how **write.csv()** function is used to create an output CSV file.

Example

```
1. csv_data<- read.csv("record.csv")
2.
3. #Getting details of those peoples who joined on or after 2014
4. details <- subset(csv_data,as.Date(start_date)>as.Date("2014-01-01"))
5.
6. # Writing filtered data into a new file.
7. write.csv(details,"output.csv")
8. new_details<- read.csv("output.csv")
9. print(new_details)
```

Output

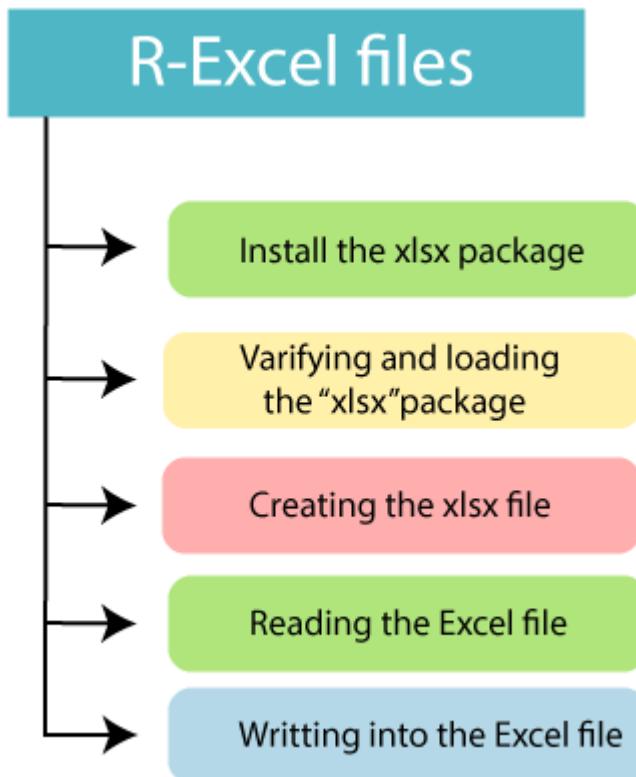


The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window contains the following text:

```
C:\Users\ajeet\R>Rscript datafile.R
  X id      name salary start_date      dept
1 3  3 Vaishali  63.00 2014-11-15      IT
2 4  4 Nishka   749.00 2014-05-11      HR
3 5  5 Gunjan   863.25 2015-03-27 Finance
4 8  8 Akash    712.50 2014-06-17 Financ

C:\Users\ajeet\R>
```

R - Excel File



Install xlsx Package

Our primary task is to install "xlsx" package with the help of `install.package` command. When we install the xlsx package, it will ask us to install some additional packages on which this package is dependent. For installing the additional packages, the same command is used with the required package name. There is the following syntax of `install` command:

First Method

Example

1. `#Installing xlsx package`
2. `install.packages("xlsx")`
3. `# Verifying the package is installed.`
4. `any(grepl("xlsx",installed.packages()))`
- 5.
6. `# Loading the library into R workspace.`
7. `library("xlsx")`

Output

```
Rterm (64-bit)
Type 'q()' to quit R.

[Previously saved workspace restored]

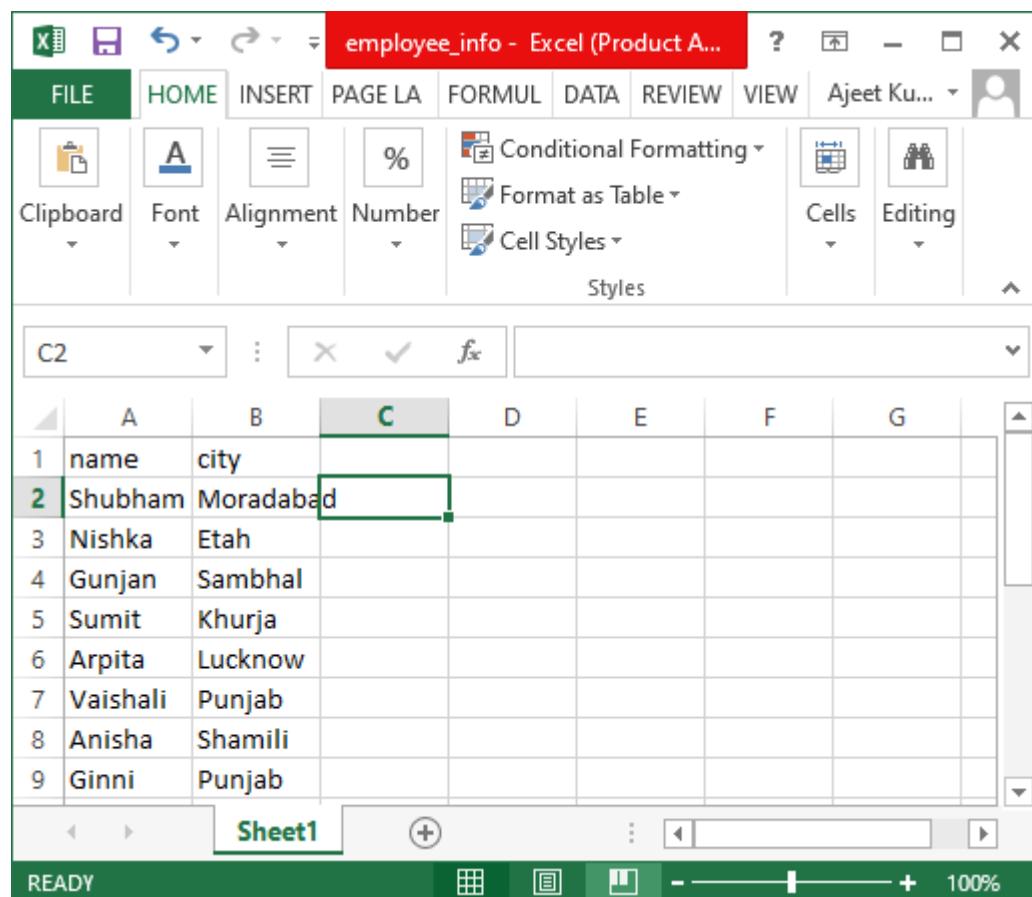
> #Installing xlsx package
> install.packages("xlsx")
Installing package into 'C:/Users/ajeet/OneDrive/Documents/R/win-library/3.6'
(as 'lib' is unspecified)
--- Please select a CRAN mirror for use in this session ---
trying URL 'https://cloud.r-project.org/bin/windows/contrib/3.6/xlsx_0.6.1.zip'
Content type 'application/zip' length 460695 bytes (449 KB)
downloaded 449 KB

package 'xlsx' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
      C:\Users\ajeet\AppData\Local\Temp\RtmpkpsFGi\downloaded_packages
>
> # Verifying the package is installed.
> any(grepl("xlsx",installed.packages()))
[1] TRUE
>
> # Loading the library into R workspace.
> library("xlsx")
> print(library("xlsx"))
Error in library("xlsx") : could not find function "library"
> print(library("xlsx"))
[1] "xlsx"      "stats"     "graphics"   "grDevices"  "utils"      "datasets"
[7] "methods"   "base"
>
```

Creating an xlsx File

Once the xlsx package is loaded into our system, we will create an excel file with the following data and named it employee.



The screenshot shows a Microsoft Excel window titled "employee_info - Excel (Product A...)" with the "HOME" tab selected. The ribbon also includes FILE, INSERT, PAGE LA, FORMUL, DATA, REVIEW, and VIEW tabs. The user profile "Ajeet Ku..." is visible on the right. The Excel interface features the following elements:

- Clipboard:** Contains icons for Copy, Cut, Paste, and Undo/Redo.
- Font:** Includes icons for Bold, Italic, Underline, and Font Size.
- Alignment:** Includes icons for Align Left, Center, Align Right, and Wrap Text.
- Number:** Includes icons for Number Format, Percentage, and Currency.
- Styles:** Includes Conditional Formatting, Format as Table, and Cell Styles dropdowns.
- Cells:** Includes icons for Select All, Find & Select, and Sort & Filter.
- Editing:** Includes icons for Insert, Delete, and Format Painter.

The worksheet displays a table with 9 rows and 2 columns. The columns are labeled A and B. The first row contains headers "name" and "city". The subsequent rows contain data:

	A	B
1	name	city
2	Shubham	Moradabad
3	Nishka	Etah
4	Gunjan	Sambhal
5	Sumit	Khurja
6	Arpita	Lucknow
7	Vaishali	Punjab
8	Anisha	Shamili
9	Ginni	Punjab

The cell containing "Moradabad" is currently selected and highlighted with a green border. The status bar at the bottom shows "READY" and "100%".

Note: Both the files will be saved in the current working directory of the R workspace.

Reading the Excel File

Like the CSV file, we can read data from an excel file. R provides `read_xlsx` function, which takes two arguments as input, i.e., file name and index of the sheet. This function returns the excel data in the form of a data frame in the R environment. There is the following syntax of `read_xlsx()` function:

Second Method

```
install.packages("readxl")  
  
library(readxl)  
  
excel_data<- read_xlsx("employee.xlsx")  
  
excel_data  
  
View(excel_data)  
  
nrow(excel_data)  
  
ncol(excel_data)  
  
colnames(excel_data)  
  
str(excel_data)  
  
data2<-read_excel("employee.xlsx",sheet=1)  
  
View(data2)  
  
data3<-read_excel("employee.xlsx",sheet="sheetname")  
  
View(data3)  
  
#choose by drive  
  
excel_data<-read_excel(file.choose())  
  
View(excel_data)  
  
str(excel_data)
```

R - Binary Files

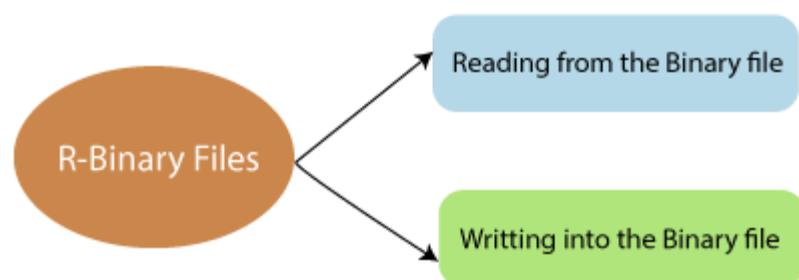
R Binary File

A binary file is a file which contains information present only in the form of bits and bytes(0's and 1's). They are not human-readable because the bytes translate into characters and symbols that contain many other non-printable characters. If we will read a binary file using any text editor, it will show the characters like ð and Ø.

The code is relatively very easy to read binary data into R. To read binary data, we must know how a piece of information has been parsed into binary.

The binary file must be read by specific programs to be useful. For example, the binary file of a Microsoft Word program can only be read by the Word program in a human-readable form. It indicates that, in addition to human-readable text, there is a lot of information such as character formatting and page numbers, etc., which are also stored with alphanumeric characters. And finally, a binary file is a contiguous sequence of bytes. The line break we see in a text file is a character joining the first line to the next line.

Sometimes, the data generated by other programs need to be processed by R as a binary file. Also, R needs to create binary files that can be shared with other programs. There are two functions `writeBin ()` and `readBin ()` for creating and reading binary files in R.



Writing the Binary File

Like CSV and Excel files, we can also write into a binary file. R provides a `writeBin()` function for writing the data into a binary file. There is the following syntax of `writeBin()` function:

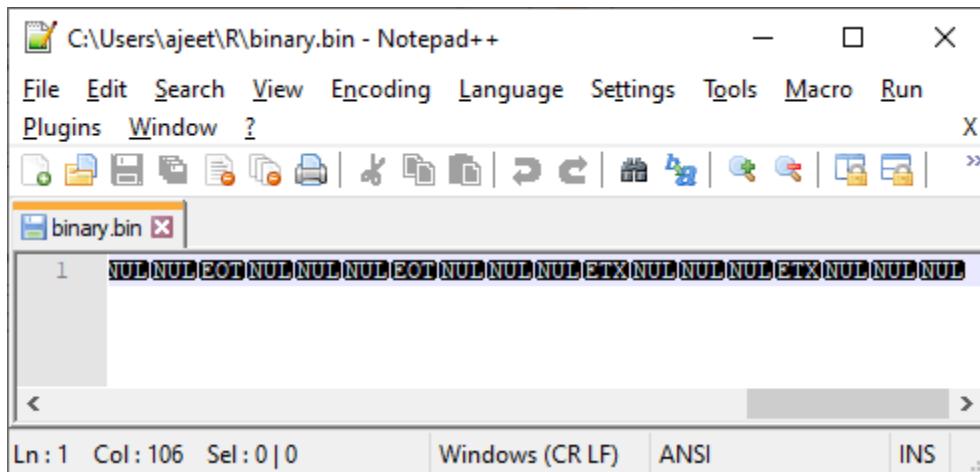
```
df = data.frame(  
  "ID" = c(1, 2, 3, 4),  
  "Name" = c("Tony", "Thor", "Loki", "Hulk"),  
  "Age" = c(20, 34, 24, 40),  
  "Pin" = c(756083, 756001, 751003, 110011)  
)
```

```
# Creating a connection object  
# to write the binary file using mode "wb"  
con = file("myfile.dat", "wb")  
  
# Write the column names of the data frame  
# to the connection object  
writeBin(colnames(df), con)
```

```
# Write the records in each of the columns to the file  
writeBin(c(df$ID, df>Name, df$Age, df$Pin), con)
```

```
# Close the connection object  
close(con)
```

Output



Reading the Binary File

We can also read our binary file which we have created before. For this purpose, R provides a `readBin()` function for reading the data from a binary file.

There is the following syntax of `readbin()` function:

```
con = file("myfile.dat", "rb")
```

```
# Read the column names
```

```
# n = 4 as here 4 column
```

```
colname = readBin(con, character(), n = 4)
```

```
# Read column values
```

```
# n = 20 as here 16 values and 4 column names
```

```
con = file("myfile.dat", "rb")
```

```
bindata = readBin(con, integer(), n = 20)
```

```
# Read the ID values
```

```

# as first 1:4 byte for col name

# then values of ID col is within 5 to 8

ID = bindata[5:8]

# Similarly 9 to 12 byte for values of name column

Name = bindata[9:12]

# 13 to 16 byte for values of the age column

Age = bindata[13:16]

# 17 to 20 byte for values of Pincode column

PinCode = bindata[17:20]

# Combining all the values and make it a data frame

finaldata = cbind(ID, Name, Age, PinCode)

colnames(finaldata)= colname

print(finaldata)

```

Output

```

> finaldata
      ID      Name     Age      Pin
[1,] 3276849 1867251826 875692084 808465973
[2,] 3407923 1207986539 3159040 892796977
[3,] 2037280596    7040117 808858935 858796081
[4,] 1869108224  855650354 922759992 808530176
> |

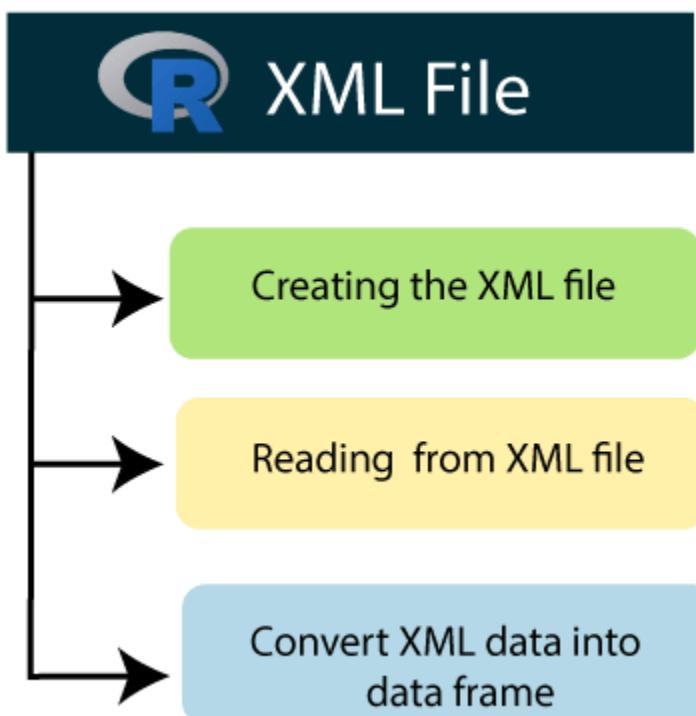
```

R - XML Files

R XML File

Like HTML, XML is also a markup language which stands for Extensible Markup Language. It is developed by World Wide Web Consortium(W3C) to define the syntax for encoding documents which both humans and machine can read. This file contains markup tags. There is a difference between HTML and XML. In HTML, the markup tag describes the structure of the page, and in xml, it describes the meaning of the data contained in the file. In R, we can read the xml files by installing "XML" package into the R environment. This package will be installed with the help of the familiar command i.e., `install.packages`.

1. `install.packages("XML")`



Creating XML File

We will create an xml file with the help of the given data. We will save the following data with the .xml file extension to create an xml file. XML tags describe the meaning of data, so that data contained in such tags can easily tell or explain about the data.

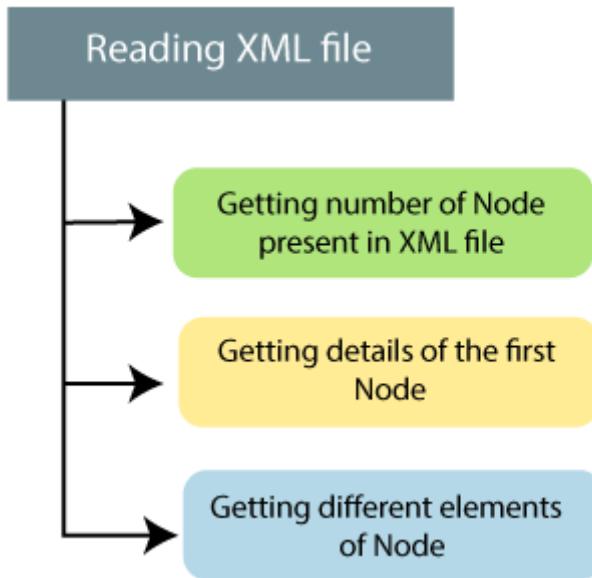
Example: xml_data.xml

```
<RECORDS>
  <STUDENT>
    <ID>1</ID>
    <NAME>Alia</NAME>
    <MARKS>620</MARKS>
    <BRANCH>IT</BRANCH>
  </STUDENT>
  <STUDENT>
    <ID>2</ID>
    <NAME>Brijesh</NAME>
    <MARKS>440</MARKS>
    <BRANCH>Commerce</BRANCH>
  </STUDENT>
  <STUDENT>
    <ID>3</ID>
    <NAME>Yash</NAME>
    <MARKS>600</MARKS>
    <BRANCH>Humanities</BRANCH>
  </STUDENT>
  <STUDENT>
    <ID>4</ID>
    <NAME>Mallika</NAME>
    <MARKS>660</MARKS>
    <BRANCH>IT</BRANCH>
  </STUDENT>
  <STUDENT>
    <ID>5</ID>
    <NAME>Zayn</NAME>
    <MARKS>560</MARKS>
    <BRANCH>IT</BRANCH>
  </STUDENT>
</RECORDS>
```

Reading XML File

In R, we can easily read an xml file with the help of `xmlParse()` function. This function is stored as a list in R. To use this function, we first need to load the `xml` package with the help of the `library()` function. Apart from the `xml` package, we also need to load one additional package named `methods`.

Let's see an example to understand the working of `xmlParse()` function in which we read our `xml_data.xml` file.



Example: Reading xml data in the form of a list.

```

library("XML")
library("methods")

# the contents of xml_data.xml are parsed
data <- xmlParse(file = "xml_data.xml")

print(data)
  
```

Output

```

<?xml version="1.0"?>
<RECORDS>
  <STUDENT>
    <ID>1</ID>
    <NAME>Alia</NAME>
    <MARKS>620</MARKS>
    <BRANCH>IT</BRANCH>
  </STUDENT>
  
```

Example: Getting number of nodes present in xml file.

How to convert xml data into a data frame

It's not easy to handle data effectively in large files. For this purpose, we read the data in the xml file as a data frame. Then this data frame is processed by the data analyst. R provide `xmlToDataFrame()` function to extract the information in the form of Data Frame.

Let's see an example to understand how this function is used and processed:

Example

```
library("XML")
library("methods")

# Convert the input xml file to a data frame.

dataframe <- xmlToDataFrame("xml_data.xml")

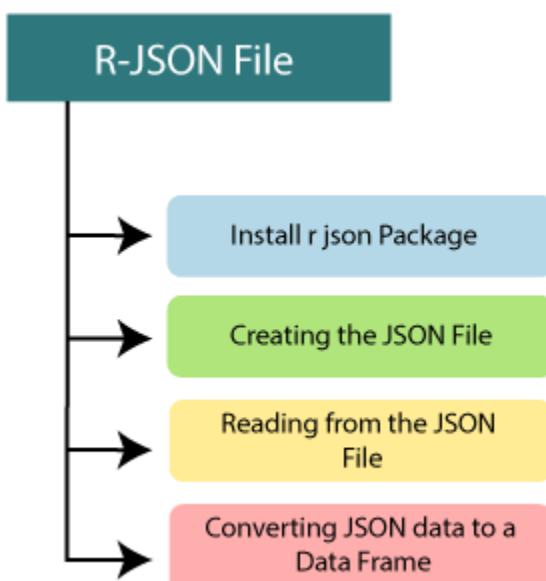
print(dataframe)
```

Output

ID	NAME	MARKS	BRANCH
1	1	Alia	620
2	2	Brijesh	440
3	3	Yash	600
4	4	Mallika	660
5	5	Zayn	560

R JSON File

JSON stands for JavaScript Object Notation. The JSON file contains the data as text in a human-readable format. Like other files, we can also read and write into the JSON files. For this purpose, R provides a package named rjson, which we have to install with the help of the familiar command **install.packages**.



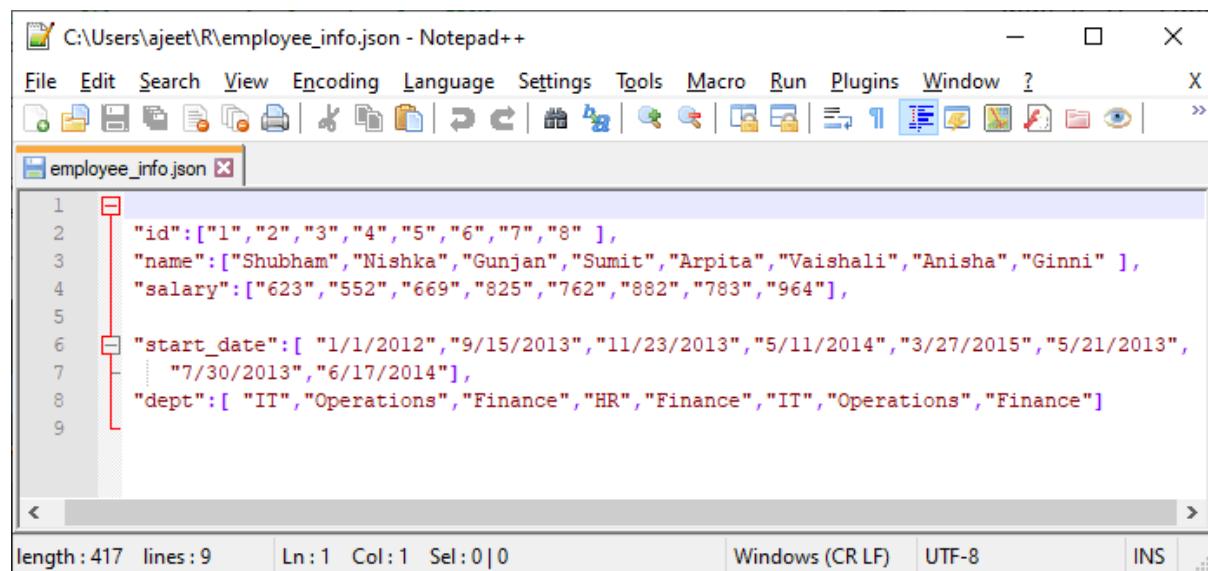
Creating a JSON file

The extension of JSON file is .json. To create the JSON file, we will save the following data as employee_info.json. We can write the information of employees in any text editor with its appropriate rule of writing the JSON file. In JSON files, the information contains in between the curly braces({}).

Example: employee.json

```
{  
    "ID": ["1", "2", "3", "4", "5"],  
    "Name": ["Mithuna", "Tanushree", "Parnasha", "Arjun", "Pankaj"],  
    "Salary": ["722.5", "815.2", "1611", "2829", "843.25"],  
  
    "StartDate": ["6/17/2014", "1/1/2012", "11/15/2014", "9/23/2013", "5/21/  
2013"],  
    "Dept": ["IT", "IT", "HR", "Operations", "Finance"]  
}
```

Output



The screenshot shows the Notepad++ application window with the file 'employee_info.json' open. The JSON data is displayed in the editor:

```
1  "id": ["1", "2", "3", "4", "5", "6", "7", "8" ],  
2  "name": ["Shubham", "Nishka", "Gunjan", "Sumit", "Arpita", "Vaishali", "Anisha", "Ginni" ],  
3  "salary": ["623", "552", "669", "825", "762", "882", "783", "964"],  
4  
5  "start_date": [ "1/1/2012", "9/15/2013", "11/23/2013", "5/11/2014", "3/27/2015", "5/21/2013",  
6      "7/30/2013", "6/17/2014"],  
7  "dept": [ "IT", "Operations", "Finance", "HR", "Finance", "IT", "Operations", "Finance" ]  
8  
9
```

The status bar at the bottom shows: length : 417 lines : 9 Ln:1 Col:1 Sel:0|0 Windows (CR LF) UTF-8 INS

Install rjson package

By running the following command into the R console, we will install the rjson package into our current working directory.

```
# Read a JSON file

# Load the package required to read JSON files.
library("rjson")

# Give the input file name to the function.
result <- fromJSON(file = "example.json")

# Print the result.
print(result)
```

Output

```
$ID
[1] "1" "2" "3" "4" "5"

$Name
[1] "Mithuna"    "Tanushree"  "Parnasha"   "Arjun"      "Pankaj"

$Salary
[1] "722.5"     "815.2"     "1611"       "2829"      "843.25"

$StartDate
[1] "6/17/2014"  "1/1/2012"   "11/15/2014" "9/23/2013" "5/21/2013"

$Dept
[1] "IT"          "IT"         "HR"         "Operations" "Finance"
```

Converting JSON data to a Data Frame

R provide, as.data.frame() function to convert the extracted data into data frame. For further analysis, data analysts use this function. Let's start an example to see how this function is used, and in our example, we will consider our employee_info.json file.

Example

```
# Convert the file into dataframe
# Load the package required to read JSON files.

library("rjson")
```

```

# Give the input file name to the function.

result <- fromJSON(file = "example.json")

# Convert JSON file to a data frame.

json_data_frame <- as.data.frame(result)

print(json_data_frame)

```

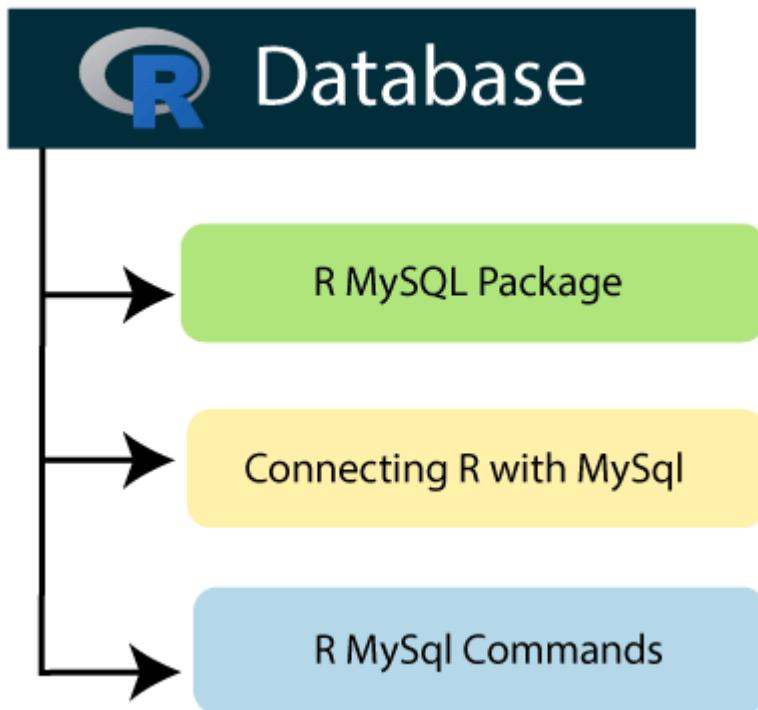
Output

ID	Name	Salary	StartDate	Dept
1	Mithuna	722.5	6/17/2014	IT
2	Tanushree	815.2	1/1/2012	IT
3	Parnasha	1611	11/15/2014	HR
4	Arjun	2829	9/23/2013	Operations
5	Pankaj	843.25	5/21/2013	Finance

R Database

In the relational database management system, the data is stored in a normalized format. Therefore, to complete statistical computing, we need very advanced and complex SQL queries. The large and huge data which is present in the form of tables require SQL queries to extract the data from it.

R can easily connect with many of the relational databases like MySql, SQL Server, Oracle, etc. When we extract the information from these databases, by default, the information is extracted in the form of data frame. Once, the data comes from the database to the R environment; it will become a normal R dataset. The data analyst can easily analyze or manipulate the data with the help of all the powerful packages and functions.



RMySQL Package

RMySQL package is one of the most important built-in package of R. This package provides native connectivity between the R and MySql database. In R, to work with MySql database, we first have to install the RMySQL package with the help of the familiar command, which is as follows:

```
# Install package  
install.packages("RMySQL")  
  
# Loading library  
library("RMySQL")  
  
# Create connection  
mysqlconn = dbConnect(MySQL(), user = 'root', password = 'root',  
                      dbname = 'sachin', host = 'localhost')  
  
# Show tables in database  
dbListTables(mysqlconn)
```

Output

We have created a database employee in which there is a table employee_info, which has the following record.

The screenshot shows the phpMyAdmin interface for a MySQL database named 'nodedatabase'. The 'student' table is selected. The SQL query 'SELECT * FROM `student`' is run, and the result is displayed in a table:

Rollno	Name	Marks
101	James	70

We will use the data which we have mentioned above in our upcoming topics.

```
# Select all rows from articles table
res = dbSendQuery(mysqlconn, "SELECT *FROM student")

# Fetch first 3 rows in data frame
df = fetch(res, n = 3)
print(df)
```

Rollno	Name	Marks
101	James	70

R MySQL Commands

In R, we can perform all the SQL commands like insert, delete, update, etc. For performing the query on the database, R provides the `dbSendQuery()` function. The query is executed in MySQL, and the result set is returned using the R `fetch()` function. Finally, it is stored in R as a data frame. Let's see the example of each and every SQL command to understand how `dbSendQuery()` and `fetch()` functions are used.



Create Table

R provides an additional function to create a table into the database i.e., dbWriteTable(). This function creates a table in the database; if it does not exist else, it will overwrite the table. This function takes the data frame as an input.

Example

1. #Loading RMySQL package into R
2. library("RMySQL")
- 3.
4. # Creating a connection Object to MySQL database.
5. # Conneting with database named "employee" which we have created befoe with the helpof XAMPP server.
6. `mysql_connect = dbConnect(MySQL(), user = 'root', password = "", dbname = 'employee', host = 'localhost')`
- 8.
9. #Creating data frame to create a table
10. `emp.data <- data.frame(`
11. `name = c("Raman","Rafia","Himanshu","jasmine","Yash"),`

```

12. salary = c(623.3,915.2,611.0,729.0,843.25),
13. start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-
11","2015-03-27")),
14. dept = c("Operations","IT","HR","IT","Finance"),
15. stringsAsFactors = FALSE
16. )
17.
18. # All the rows of emp.data are taken into MySql.
19. dbWriteTable(mysql_connect, "emp", emp.data[, ], overwrite = TRUE)

```

Output

The screenshot shows two windows. The top window is a Command Prompt titled 'Command Prompt' with the following text:

```
C:\Users\ajeet\R>Rscript database.R
Loading required package: DBI
[1] TRUE

C:\Users\ajeet\R>
```

The bottom window is a browser window titled 'localhost / 127.0.0.1 / employee /' displaying the phpMyAdmin interface. The 'Structure' tab is selected, showing a table named 'emp' with columns: row_names, name, salary, start_date, and dept. The data is as follows:

row_names	name	salary	start_date	dept
1	Raman	623.3	2012-01-01	Operations
2	Rafia	915.2	2013-09-23	IT
3	Himanshu	611	2014-11-15	HR
4	jasmine	729	2014-05-11	IT
5	Yash	843.25	2015-03-27	Finance

Select

We can simply select the record from the table with the help of the `fetch()` and `dbSendQuery()` function. Let's see an example to understand how to select query works with these two functions.

Example

```
1. #Loading RMySQL package into R
2. library("RMySQL")
3.
4. # Creating a connection Object to MySQL database.
5. # Conneting with database named "employee" which we have created befoe
   with the helpof XAMPP server.
6. mysql_connect = dbConnect(MySQL(), user = 'root', password = "", dbname = 'emplo
   yee',
7.   host = 'localhost')
8.
9. # selecting the record from employee_info table.
10. record = dbSendQuery(mysql_connect, "select * from employee_info")
11.
12. # Storing the result in a R data frame object. n = 6 is used to fetch first 6 rows.
13. data_frame = fetch(record, n = 6)
14. print(data_frame)
```

Output

```
C:\Users\ajeet\R>Rscript database.R
Loading required package: DBI
  id      name    salary      date     dept
1  1    Shubham     623  1/1/2012       IT
2  2     Nishka     552  9/23/2012 Operations
3  3    Gunjan     669 11/15/2014       IT
4  4     Sumit     825  5/11/2014       HR
5  5    Arpita     762  3/27/2012 Finance
6  6   Vaishali     882  5/21/2013       IT
```

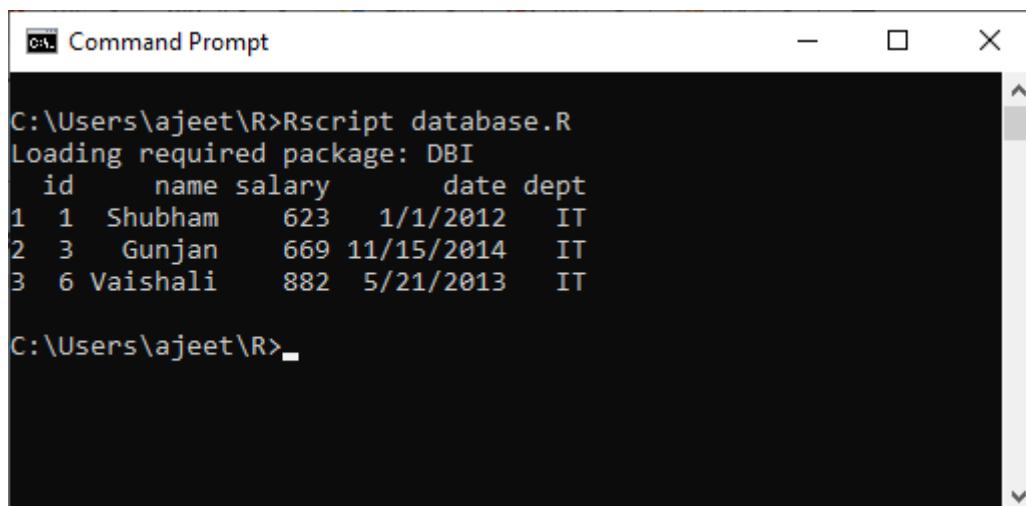
Select with where clause

We can select the specific record from the table with the help of the `fetch()` and `dbSendQuery()` function. Let's see an example to understand how to select query works with where clause and these two functions.

Example

```
1. #Loading RMySQL package into R
2. library("RMySQL")
3.
4. # Creating a connection Object to MySQL database.
5. # Conneting with database named "employee" which we have created before
   with the helpof XAMPP server.
6. mysql_connect = dbConnect(MySQL(), user = 'root', password = "", dbname = 'emplo
   yee',
7.   host = 'localhost')
8.
9. # selecting the specific record from employee_info table.
10. record = dbSendQuery(mysql_connect, "select * from employee_info where dept='IT'
    ")
11.
12. # Fetching all the records(with n = -1) and storing it as a data frame.
13. data_frame = fetch(record, n = -1)
14. print(data_frame)
```

Output



```
C:\Users\ajeet\R>Rscript database.R
Loading required package: DBI
  id      name salary      date dept
1  1 Shubham     623  1/1/2012   IT
2  3 Gunjan      669 11/15/2014   IT
3  6 Vaishali    882  5/21/2013   IT

C:\Users\ajeet\R>
```

Insert command

We can insert the data into tables with the help of the familiar method dbSendQuery() function.

Example

```
1. #Loading RMySQL package into R
```

```

2. library("RMySQL")
3.
4. # Creating a connection Object to MySQL database.
5. # Conneting with database named "employee" which we have created befoe
   with the helpof XAMPP server.
6. mysql_connect = dbConnect(MySQL(), user = 'root', password = "", dbname = 'emplo
   yee',
7.   host = 'localhost')
8.
9. # Inserting record into employee_info table.
10. dbSendQuery(mysql_connect, "insert into employee_info values(9,'Preeti',1025,'8/25/
   2013','Operations')")

```

Output

The screenshot displays two windows illustrating the process of inserting data into a MySQL database from an R script.

Top Window (Command Prompt):

```

C:\Users\ajeet\R>Rscript database.R
Loading required package: DBI
<MySQLResult:NA,0,0>
C:\Users\ajeet\R>

```

Bottom Window (phpMyAdmin):

The phpMyAdmin interface shows the 'employee' database with the 'employee_info' table selected. The table data is as follows:

id	name	salary	date	dept
1	Shubham	623	1/1/2012	IT
2	Nishka	552	9/23/2012	Operations
3	Gunjan	669	11/15/2014	IT
4	Sumit	825	5/11/2014	HR
5	Arpita	762	3/27/2012	Finance
6	Vaishali	882	5/21/2013	IT
7	Anisha	783	7/30/2013	Operations
8	Ginni	964	6/17/2013	Finance
9	Preeti	1025	8/25/2013	Operations

The last row, corresponding to the inserted record, is highlighted with a red border.

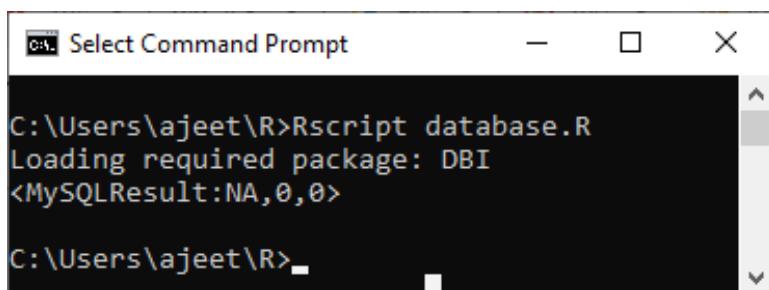
Update command

Updating a record in the table is much easier. For this purpose, we have to pass the update query to the dbSendQuery() function.

Example

1. #Loading RMySQL package into R
2. library("RMySQL")
- 3.
4. # Creating a connection Object to MySQL database.
5. # Connecting with database named "employee" which we have created before with the help of XAMPP server.
6. mysql_connect = dbConnect(MySQL(), user = 'root', password = "", dbname = 'employee',
host = 'localhost')
- 7.
- 8.
9. # Updating the record in employee_info table.
10. dbSendQuery(mysql_connect, "update employee_info set dept='IT' where id=9")

Output



```
C:\Users\ajeet\R>Rscript database.R
Loading required package: DBI
<MySQLResult:NA,0,0>
C:\Users\ajeet\R>
```

The screenshot shows the MySQL Workbench interface with the following details:

- Address Bar:** localhost / 127.0.0.1 / employee
- Toolbar:** Apps, Other bookmarks
- Table Structure:** Server: 127.0.0.1 » Database: employee » Table: employee_info
- Table Headers:** id, name, salary, date, dept
- Table Data:**

	id	name	salary	date	dept
1	Shubham	623	1/1/2012	IT	
2	Nishka	552	9/23/2012	Operations	
3	Gunjan	669	11/15/2014	IT	
4	Sumit	825	5/11/2014	HR	
5	Arpita	762	3/27/2012	Finance	
6	Vaishali	882	5/21/2013	IT	
7	Anisha	783	7/30/2013	Operations	
8	Ginni	964	6/17/2013	Finance	
9	Preeti	1025	8/25/2013	IT	
- Bottom Buttons:** Query results operations, Console, Copy to clipboard, Export, Display chart, Create view

Delete command

Below is an example in which we delete a specific row from the table by passing the delete query in the dbSendQuery() function.

Example

1. #Loading RMySQL package into R
2. library("RMySQL")
- 3.
4. # Creating a connection Object to MySQL database.
5. # Connecting with database named "employee" which we have created before with the help of XAMPP server.
6. mysql_connect = dbConnect(MySQL(), user = 'root', password = "", dbname = 'employee',
host = 'localhost')
- 7.
- 8.

9. # Deleting the specific record from employee_info table.
10. dbSendQuery(mysql_connect, "delete from employee_info where id=8")

Output

The screenshot shows two windows. The top window is a Command Prompt showing the execution of R code to delete a row from a MySQL database. The bottom window is a MySQL Workbench interface displaying the 'employee_info' table. A red box highlights the row with id=8, which has been deleted, and the text 'Delete record of id =8' is overlaid on the table area.

id	name	salary	date	dept
1	Shubham	623	1/1/2012	IT
2	Nishka	552	9/23/2012	Operations
3	Gunjan	669	11/15/2014	IT
4	Sumit	825	5/11/2014	HR
5	Arpita	762	3/27/2012	Finance
6	Vaishali	882	5/21/2013	IT
7	Anisha	783	7/30/2013	Operations
9	Preeti	1025	8/25/2013	IT

Drop command

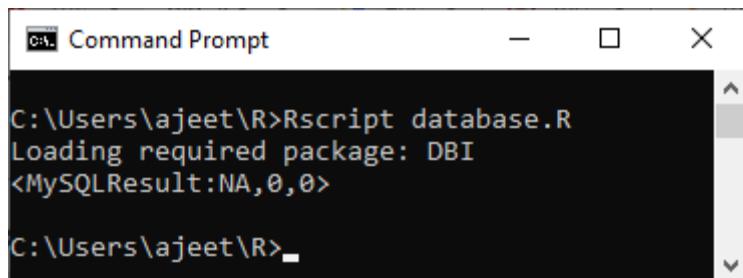
Below is an example in which we drop a table from the database by passing the appropriate drop query in the dbSendQuery() function.

Example

1. #Loading RMySQL package into R

```
2. library("RMySQL")
3.
4. # Creating a connection Object to MySQL database.
5. # Conneting with database named "employee" which we have created befoe
   with the helpof XAMPP server.
6. mysql_connect = dbConnect(MySQL(), user = 'root', password = "", dbname = 'emplo
   yee',
7.   host = 'localhost')
8.
9. # Dropping the specific table from the employee database.
10. dbSendQuery(mysql_connect, "drop table if exists emp")
```

Output



A screenshot of a Windows Command Prompt window titled "Command Prompt". The window shows the following text output:

```
C:\Users\ajeet\R>Rscript database.R
Loading required package: DBI
<MySQLResult:NA,0,0>
C:\Users\ajeet\R>
```

The screenshot shows the phpMyAdmin interface. On the left, the database tree is visible with the 'employee' database selected. Inside 'employee', there is a table named 'employee_info' which is highlighted with a red box. The main panel displays the contents of the 'employee_info' table. The table has columns: id, name, salary, date, and dept. The data is as follows:

	id	name	salary	date	dept
1	Shubham	623	1/1/2012	IT	
2	Nishka	552	9/23/2012	Operations	
3	Gunjan	669	11/15/2014	IT	
4	Sumit	825	5/11/2014	HR	
5	Arpita	762	3/27/2012	Finance	
6	Vaishali	882	5/21/2013	IT	
7	Anisha	783	7/30/2013	Operations	
9	Preeti	1025	8/25/2013	IT	

R - Web Data

Many websites provide data for consumption by its users. For example the World Health Organization(WHO) provides reports on health and medical information in the form of CSV, txt and XML files. Using R programs, we can programmatically extract specific data from such websites. Some packages in R which are used to scrap data from the web are – "RCurl", "XML", and "stringr". They are used to connect to the URL's, identify required links for the files and download them to the local environment.

Install R Packages

The following packages are required for processing the URL's and links to the files. If they are not available in your R Environment, you can install them using following commands.

```
install.packages("RCurl")
install.packages("XML")
install.packages("stringr")
install.packages("plyr")
```

Input Data

We will visit the URL [weather data](#) and download the CSV files using R for the year 2015.

Example

We will use the function **getHTMLLinks()** to gather the URLs of the files. Then we will use the function **download.file()** to save the files to the local system. As we will be applying the same code again and again for multiple files, we will create a function to be called multiple times. The filenames are passed as parameters in form of a R list object to this function.

```
# Read the URL.  
url <- "http://www.geos.ed.ac.uk/~weather/jcmb_ws/"  
  
# Gather the html links present in the webpage.  
links <- getHTMLLinks(url)  
  
# Identify only the links which point to the JCMB 2015 files.  
filenames <- links[str_detect(links, "JCMB_2015")]  
  
# Store the file names as a list.  
filenames_list <- as.list(filenames)  
  
# Create a function to download the files by passing the URL and filename list.  
downloadcsv <- function (mainurl,filename) {  
  filedetails <- str_c(mainurl,filename)  
  download.file(filedetails,filename)  
}  
  
# Now apply the l_ply function and save the files into the current R working directory.  
l_ply(filenames,downloadcsv,mainurl = "http://www.geos.ed.ac.uk/~weather/jcmb_ws/")
```

Verify the File Download

After running the above code, you can locate the following files in the current R working directory.

```
"JCMB_2015.csv" "JCMB_2015_Apr.csv" "JCMB_2015_Feb.csv" "JCMB_2015_Jan.csv"  
"JCMB_2015_Mar.csv"
```

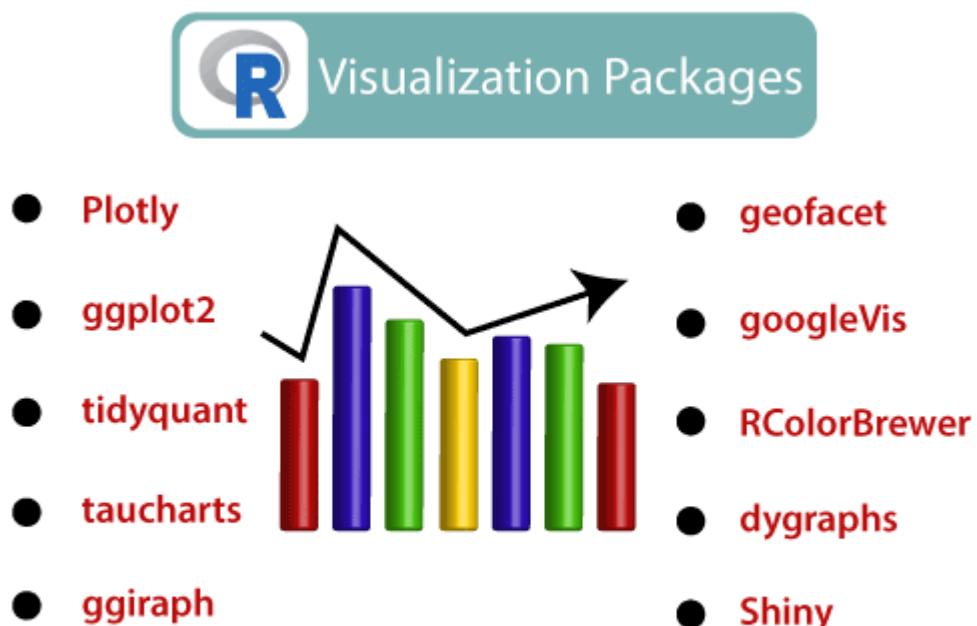
R Data Visualization

In R, we can create visually appealing data visualizations by writing few lines of code. For this purpose, we use the diverse functionalities of R. Data visualization is an efficient technique for gaining insight about data through a visual medium. With the help of visualization techniques, a human can easily obtain information about hidden patterns in data that might be neglected.

By using the data visualization technique, we can work with large datasets to efficiently obtain key insights about it.

R Visualization Packages

R provides a series of packages for data visualization. These packages are as follows:



1) **plotly**

The plotly package provides online interactive and quality graphs. This package extends upon the JavaScript library ?plotly.js.

2) **ggplot2**

R allows us to create graphics declaratively. R provides the **ggplot** package for this purpose. This package is famous for its elegant and quality graphs, which sets it apart from other visualization packages.

3) tidyquant

The **tidyquant** is a financial package that is used for carrying out quantitative financial analysis. This package adds under tidyverse universe as a financial package that is used for importing, analyzing, and visualizing the data.

4) taucharts

Data plays an important role in taucharts. The library provides a declarative interface for rapid mapping of data fields to visual properties.

5) ggiraph

It is a tool that allows us to create dynamic ggplot graphs. This package allows us to add tooltips, JavaScript actions, and animations to the graphics.

6) geofacets

This package provides geofaceting functionality for 'ggplot2'. Geofaceting arranges a sequence of plots for different geographical entities into a grid that preserves some of the geographical orientation.

7) googleVis

googleVis provides an interface between R and Google's charts tools. With the help of this package, we can create web pages with interactive charts based on R data frames.

8) RColorBrewer

This package provides color schemes for maps and other graphics, which are designed by Cynthia Brewer.

9) dygraphs

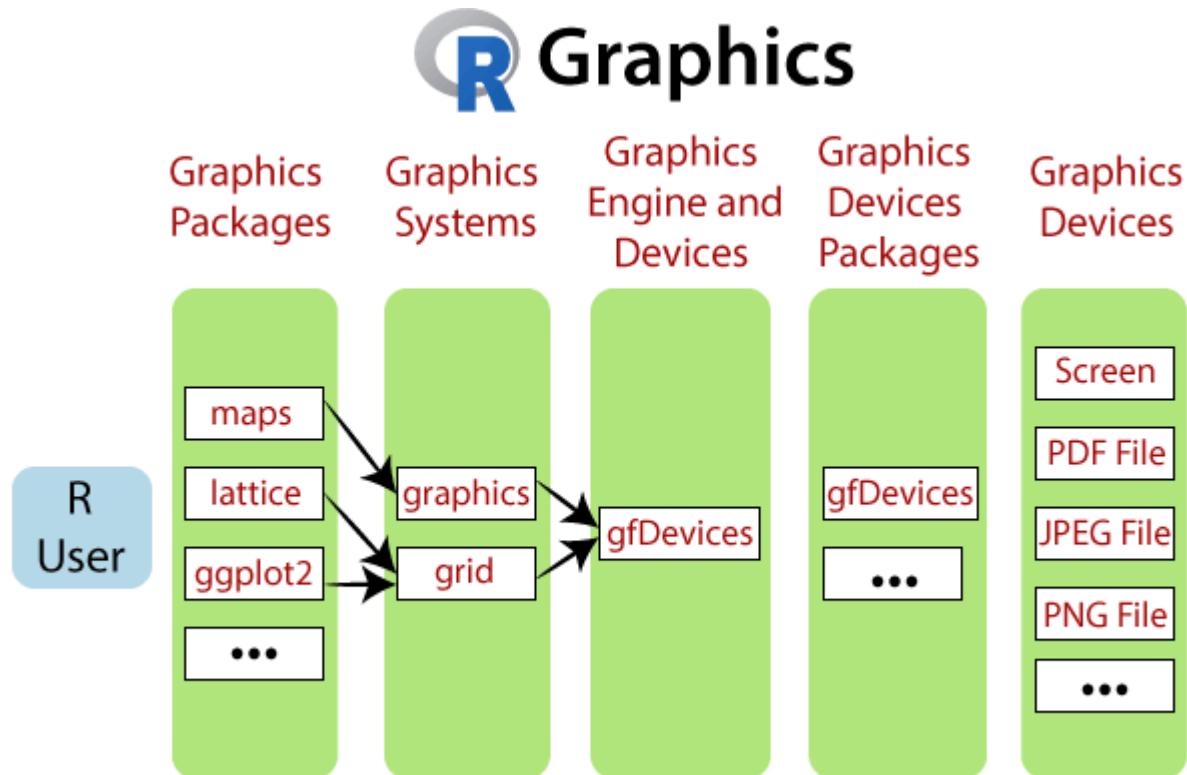
The dygraphs package is an R interface to the dygraphs JavaScript charting library. It provides rich features for charting time-series data in R.

10) shiny

R allows us to develop interactive and aesthetically pleasing web apps by providing a **shiny** package. This package provides various extensions with HTML widgets, CSS, and JavaScript.

R Graphics

Graphics play an important role in carrying out the important features of the data. Graphics are used to examine marginal distributions, relationships between variables, and summary of very large data. It is a very important complement for many statistical and computational techniques.



Standard Graphics

R standard graphics are available through package `graphics`, include several functions which provide statistical plots, like:

- Scatterplots
- Piecharts
- Boxplots
- Barplots etc.

We use the above graphs that are typically a single function call.

Graphics Devices

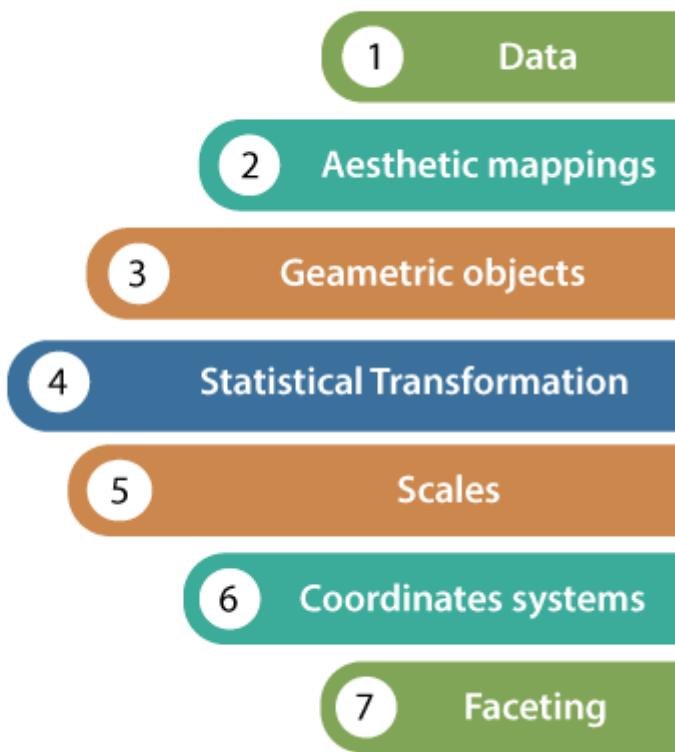
It is something where we can make a plot to appear. A graphics device is a window on your computer (screen device), a PDF file (file device), a Scalable Vector Graphics (SVG) file (file device), or a PNG or JPEG file (file device).

There are some of the following points which are essential to understand:

- The functions of graphics devices produce output, which depends on the active graphics device.
- A screen is the default and most frequently used device.
- R graphical devices such as the PDF device, the JPEG device, etc. are used.
- We just need to open the graphics output device which we want. Therefore, R takes care of producing the type of output which is required by the device.
- For producing a certain plot on the screen or as a GIF R graphics file, the R code should exactly be the same. We only need to open the target output device before.
- Several devices can be open at the same time, but there will be only one active device.

The basics of the grammar of graphics

There are some key elements of a statistical graphic. These elements are the basics of the grammar of graphics. Let's discuss each of the elements one by one to gain the basic knowledge of graphics.



1) Data

Data is the most crucial thing which is processed and generates an output.

2) Aesthetic Mappings

Aesthetic mappings are one of the most important elements of a statistical graphic. It controls the relation between graphics variables and data variables. In a scatter plot, it also helps to map the temperature variable of a data set into the X variable.

In graphics, it helps to map the species of a plant into the color of dots.

3) Geometric Objects

Geometric objects are used to express each observation by a point using the aesthetic mappings. It maps two variables in the data set into the x,y variables of the plot.

4) Statistical Transformations

Statistical transformations allow us to calculate the statistical analysis of the data in the plot. The statistical transformation uses the data and approximates it with the help of a regression line having x,y coordinates, and counts occurrences of certain values.

5) Scales

It is used to map the data values into values present in the coordinate system of the graphics device.

6) Coordinate system

The coordinate system plays an important role in the plotting of the data.

- Cartesian
- Plot

7) Faceting

Faceting is used to split the data into subgroups and draw sub-graphs for each group.

Advantages of Data Visualization in R

1. Understanding

It can be more attractive to look at the business. And, it is easier to understand through graphics and charts than a written document with text and numbers. Thus, it can attract a wider range of audiences. Also, it promotes the widespread use of business insights that come to make better decisions.

2. Efficiency

Its applications allow us to display a lot of information in a small space. Although, the decision-making process in business is inherently complex and multifunctional, displaying evaluation findings in a graph can allow companies to organize a lot of interrelated information in useful ways.

3. Location

Its app utilizing features such as Geographic Maps and GIS can be particularly relevant to wider business when the location is a very relevant factor. We will use maps to show business insights from various locations, also consider the seriousness of the issues, the reasons behind them, and working groups to address them.

Disadvantages of Data Visualization in R

1. Cost

R application development range a good amount of money. It may not be possible, especially for small companies, that many resources can be spent on purchasing them. To generate reports, many companies may employ professionals to create charts that can increase costs. Small enterprises are often operating in resource-limited settings, and are also receiving timely evaluation results that can often be of high importance.

2. Distraction

However, at times, data visualization apps create highly complex and fancy graphics-rich reports and charts, which may entice users to focus more on the form than the function. If we first add visual appeal, then the overall value of the graphic representation will be minimal. In resource-setting, it is required to understand how resources can be best used. And it is also not caught in the graphics trend without a clear purpose.

R Pie Charts

R programming language has several libraries for creating charts and graphs. A pie-chart is a representation of values in the form of slices of a circle with different colors. Slices are labeled with a description, and the numbers corresponding to each slice are also shown in the chart. However, pie charts are not recommended in the R documentation, and their characteristics are limited. The authors recommend a bar or dot plot on a pie chart because people are able to measure length more accurately than volume.

The Pie charts are created with the help of `pie()` function, which takes positive numbers as vector input. Additional parameters are used to control labels, colors, titles, etc.

There is the following syntax of the `pie()` function:

1. `pie(X, Labels, Radius, Main, Col, Clockwise)`

Here,

1. **X** is a vector that contains the numeric values used in the pie chart.
2. **Labels** are used to give the description to the slices.
3. **Radius** describes the radius of the pie chart.
4. **Main** describes the title of the chart.
5. **Col** defines the color palette.
6. **Clockwise** is a logical value that indicates the clockwise or anti-clockwise direction in which slices are drawn.

Example

#create Pie Graph

```
x <- c(20, 65, 15, 50)
```

```
labels <- c("India", "America", "Shri Lanka", "Nepal")
```

```
# Plotting the chart.
```

```
pie(x,labels)
```

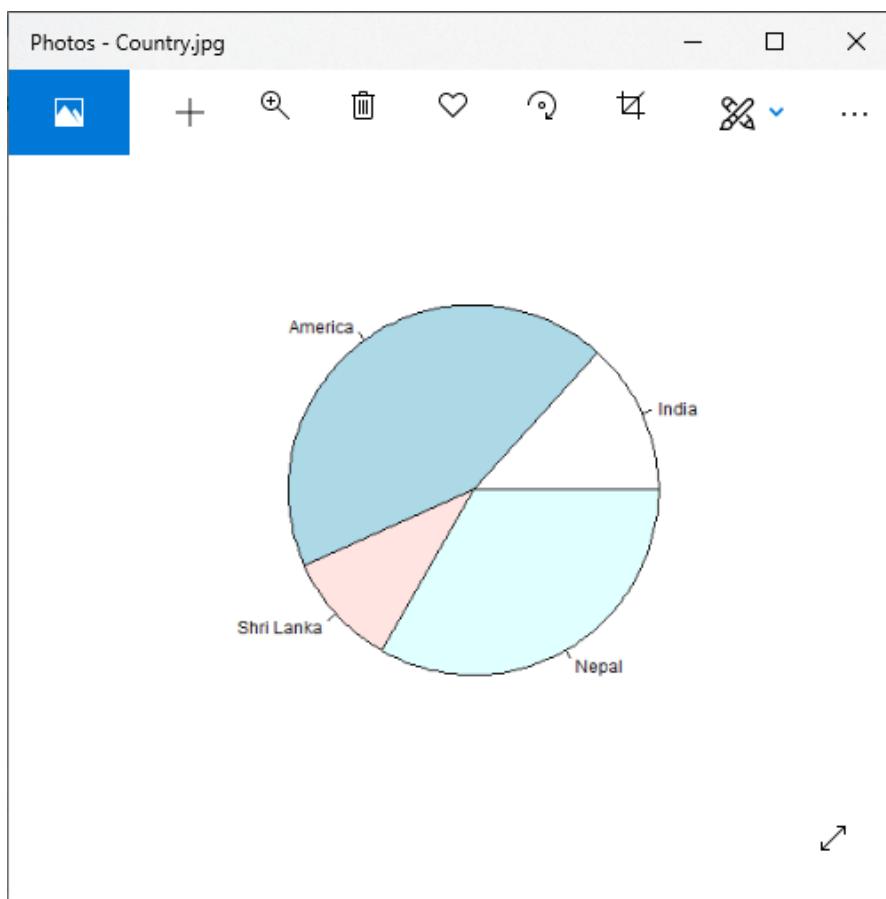
```
pie(x,labels,main = "Country Pie Chart",col=rainbow(length(x)))
```

```
#-----fill color
```

```
colors<-c("blue","green","red","orange")
```

```
pie(x,labels,main="Country Pie chart",col=colors)
```

Output:



Title and color

A pie chart has several more features that we can use by adding more parameters to the pie() function. We can give a title to our pie chart by passing the main parameter. It tells the title of the pie chart to the pie() function. Apart from this, we can use a rainbow colour pallet while drawing the chart by passing the col parameter.

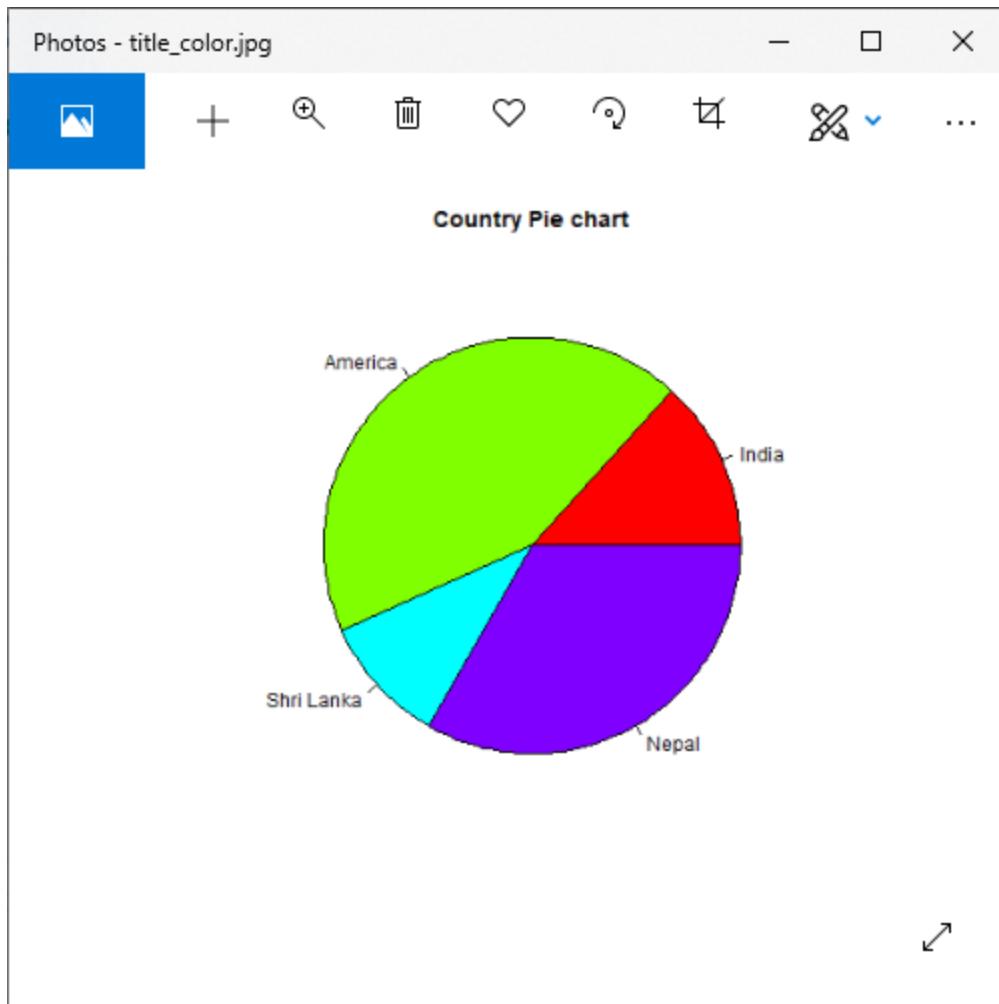
Note: *The length of the pallet will be the same as the number of values that we have for the chart. So for that, we will use length() function.*

Example

#---add legends format

```
#legend(x,y=NULL,legend,fill,col,bg)  
  
legend("topright",c("India","UK","JAPAN","USA"),cex=0.8,fill=colors)
```

Output:



Slice Percentage & Chart Legend

There are two additional properties of the pie chart, i.e., slice percentage and chart legend. We can show the data in the form of percentage as well as we can add legends to plots in R by using the legend() function. There is the following syntax of the legend() function.

1. `legend(x,y=NULL,legend,fill,col,bg)`

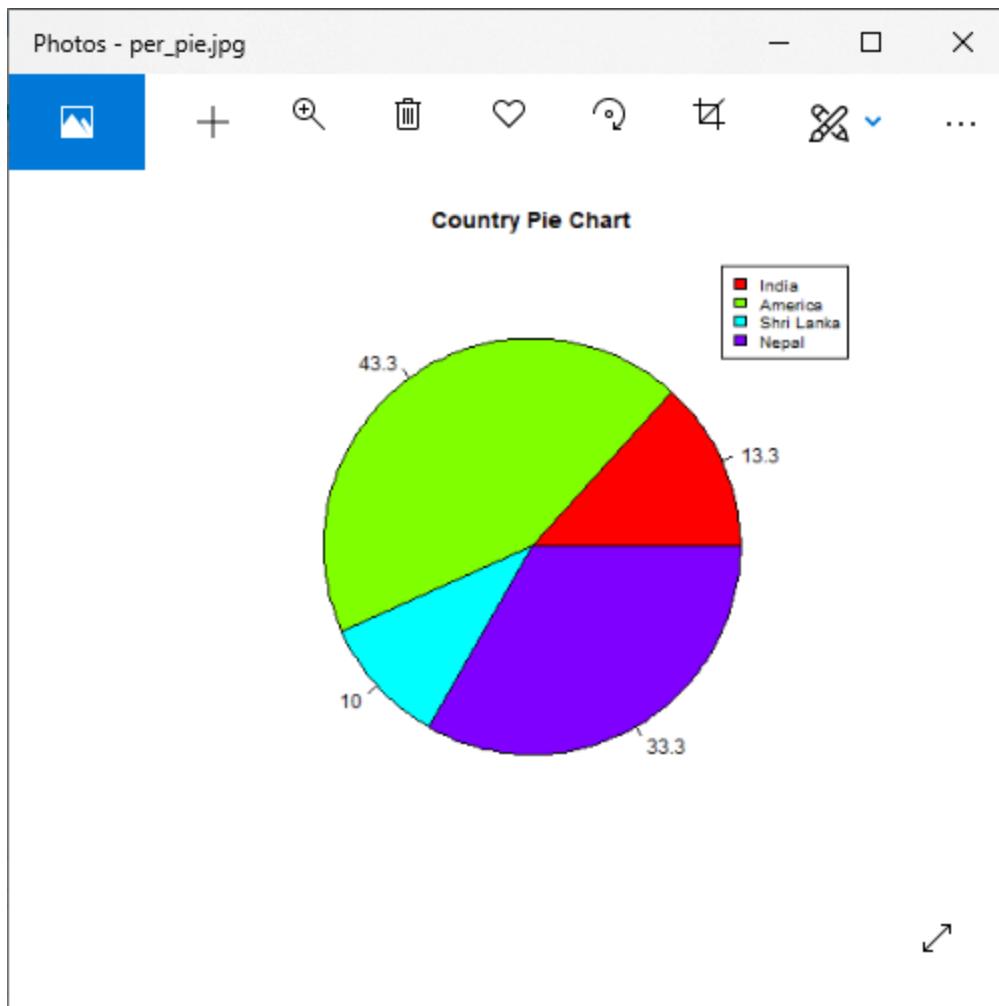
Here,

- `x` and `y` are the coordinates to be used to position the legend.
- `legend` is the text of legend
- `fill` is the color to use for filling the boxes beside the legend text.
- `col` defines the color of line and points besides the legend text.
- `bg` is the background color for the legend box.

Example

```
# Creating data for the graph.  
x <- c(20, 65, 15, 50)  
labels <- c("India", "America", "Shri Lanka", "Nepal")  
pie_percent<- round(100*x/sum(x), 1)  
# Plotting the chart.  
pie(x, labels = pie_percent, main = "Country Pie Chart", col = rainbow(length(x)))  
legend("topright", c("India", "America", "Shri Lanka", "Nepal"), cex = 0.8,  
fill = rainbow(length(x)))
```

Output:



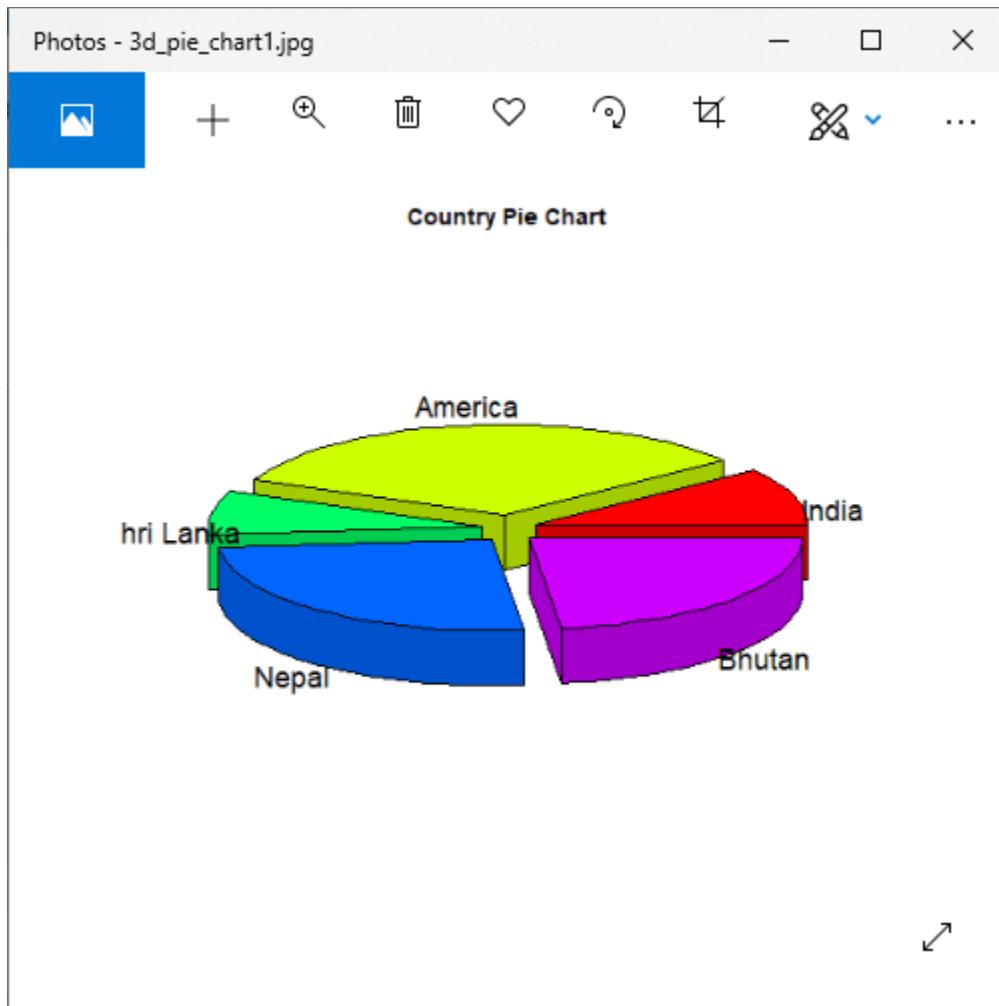
3 Dimensional Pie Chart

In R, we can also create a three-dimensional pie chart. For this purpose, R provides a `plotrix` package whose `pie3D()` function is used to create an attractive 3D pie chart. The parameters of `pie3D()` function remain same as `pie()` function. Let's see an example to understand how a 3D pie chart is created with the help of this function.

Example

```
library(plotrix)  
  
x2<-c(20,65,15,50,45)  
  
labels2<-c("India","America","Shri Lanka","Nepal","Bhutan")  
  
pie3D(x2,labels=labels2,explode=0.2,main="Country Pie Chart")
```

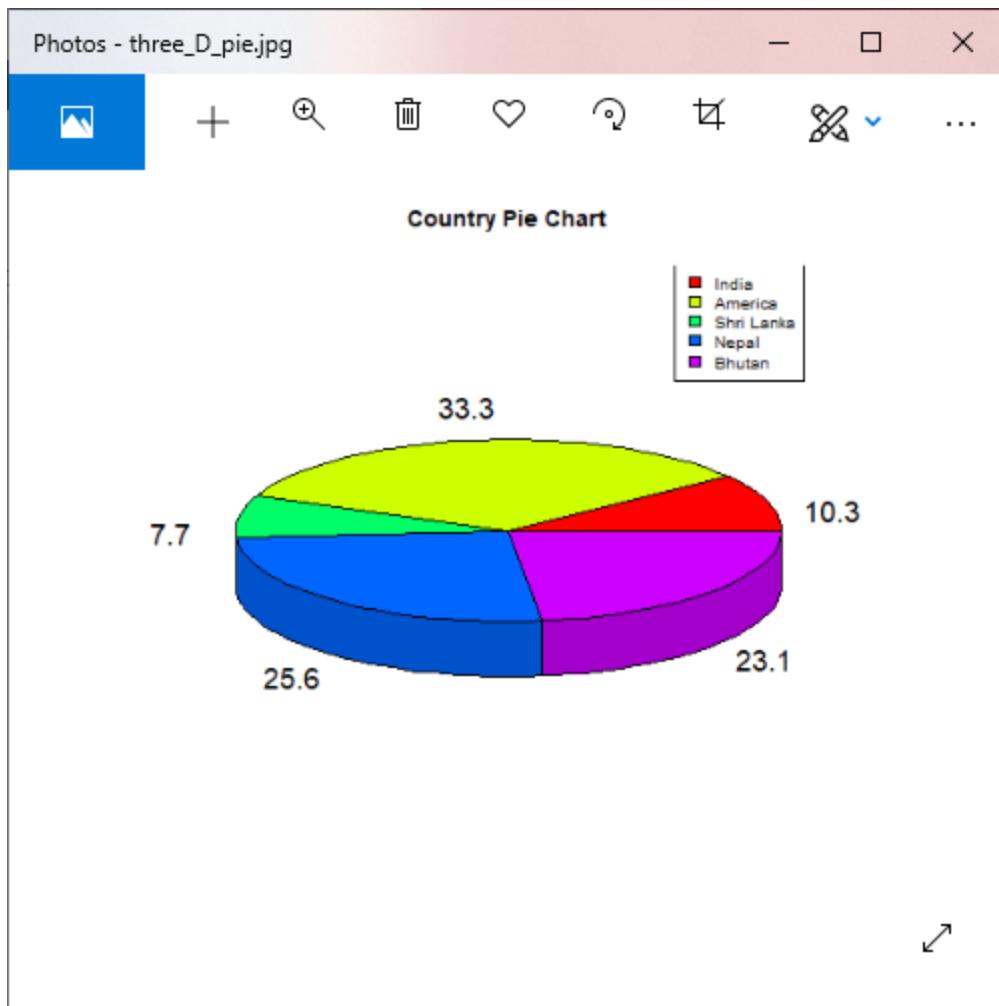
Output:



Example

```
1. # Getting the library.  
library(plotrix)  
  
# Creating data for the graph.  
x <- c(20, 65, 15, 50,45)  
labels <- c("India", "America", "Shri Lanka", "Nepal","Bhutan")  
pie_percent<- round(100*x/sum(x), 1)  
  
# Plotting the chart.  
pie3D(x, labels = pie_percent, main = "Country Pie Chart", col = rainbow(length(x)))  
legend("topright", c("India", "America", "Shri Lanka", "Nepal","Bhutan"), cex = 0.8,  
fill = rainbow(length(x)))
```

Output:



R Bar Charts

A bar chart is a pictorial representation in which numerical values of variables are represented by length or height of lines or rectangles of equal width. A bar chart is used for summarizing a set of categorical data. In bar chart, the data is shown through rectangular bars having the length of the bar proportional to the value of the variable.

In R, we can create a bar chart to visualize the data in an efficient manner. For this purpose, R provides the `barplot()` function, which has the following syntax:

1. `barplot(h,x,y,main, names.arg,col)`

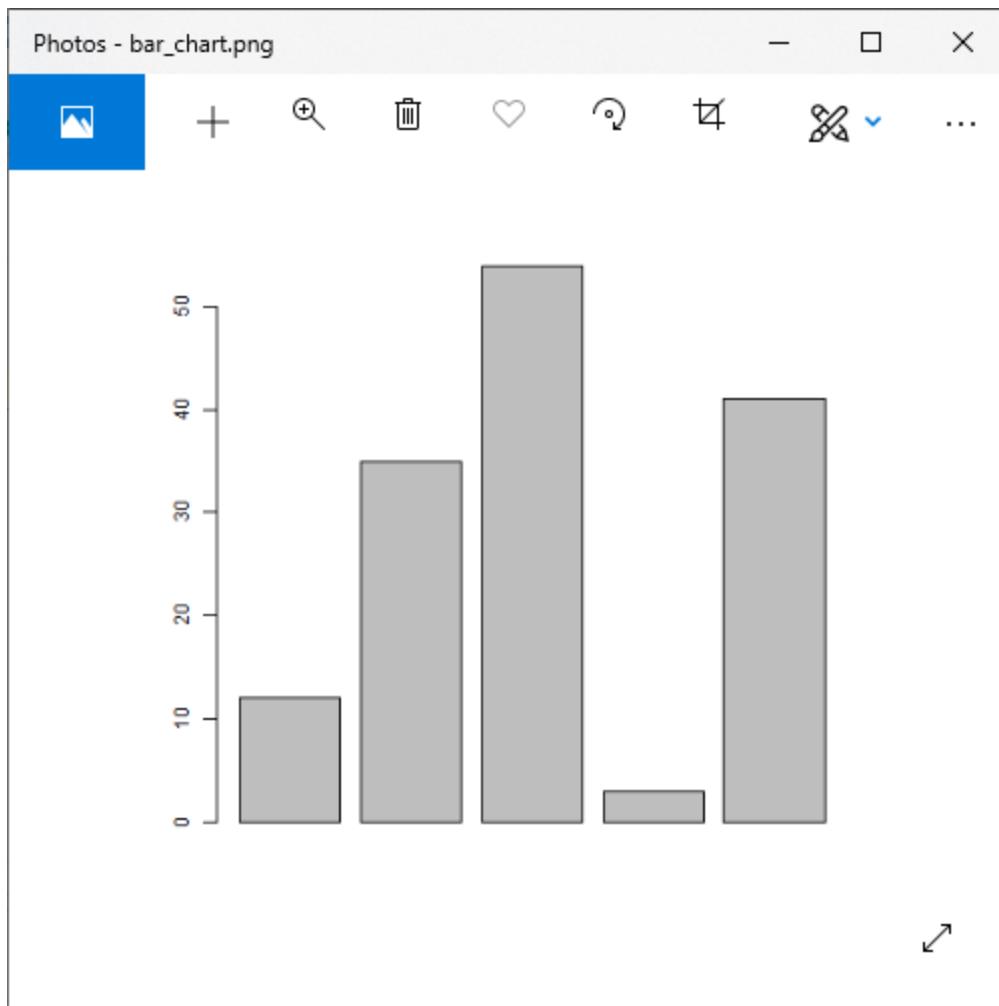
S.No	Parameter	Description
------	-----------	-------------

1.	H	A vector or matrix which contains numeric values used in the bar chart.
2.	xlab	A label for the x-axis.
3.	ylab	A label for the y-axis.
4.	main	A title of the bar chart.
5.	names.arg	A vector of names that appear under each bar.
6.	col	It is used to give colors to the bars in the graph.

Example

```
# Creating the data for Bar chart
H<- c(12,35,54,3,41)
# Plotting the bar chart
barplot(H)
```

Output:



Labels, Title & Colors

Like pie charts, we can also add more functionalities in the bar chart by-passing more arguments in the barplot() functions. We can add a title in our bar chart or can add colors to the bar by adding the main and col parameters, respectively. We can add another parameter i.e., args.name, which is a vector that has the same number of values, which are fed as the input vector to describe the meaning of each bar.

Let's see an example to understand how labels, titles, and colors are added in our bar chart.

Example

```
# Creating the data for Bar chart  
H <- c(12,35,54,3,41)  
M<- c("Feb","Mar","Apr","May","Jun")
```

```
# Plotting the bar chart  
barplot(H,names.arg=M,xlab="Month",ylab="Revenue",col="Green",  
main="Revenue Bar chart",border="red")
```

Output:



Group Bar Chart & Stacked Bar Chart

We can create bar charts with groups of bars and stacks using matrices as input values in each bar. One or more variables are represented as a matrix that is used to construct group bar charts and stacked bar charts.

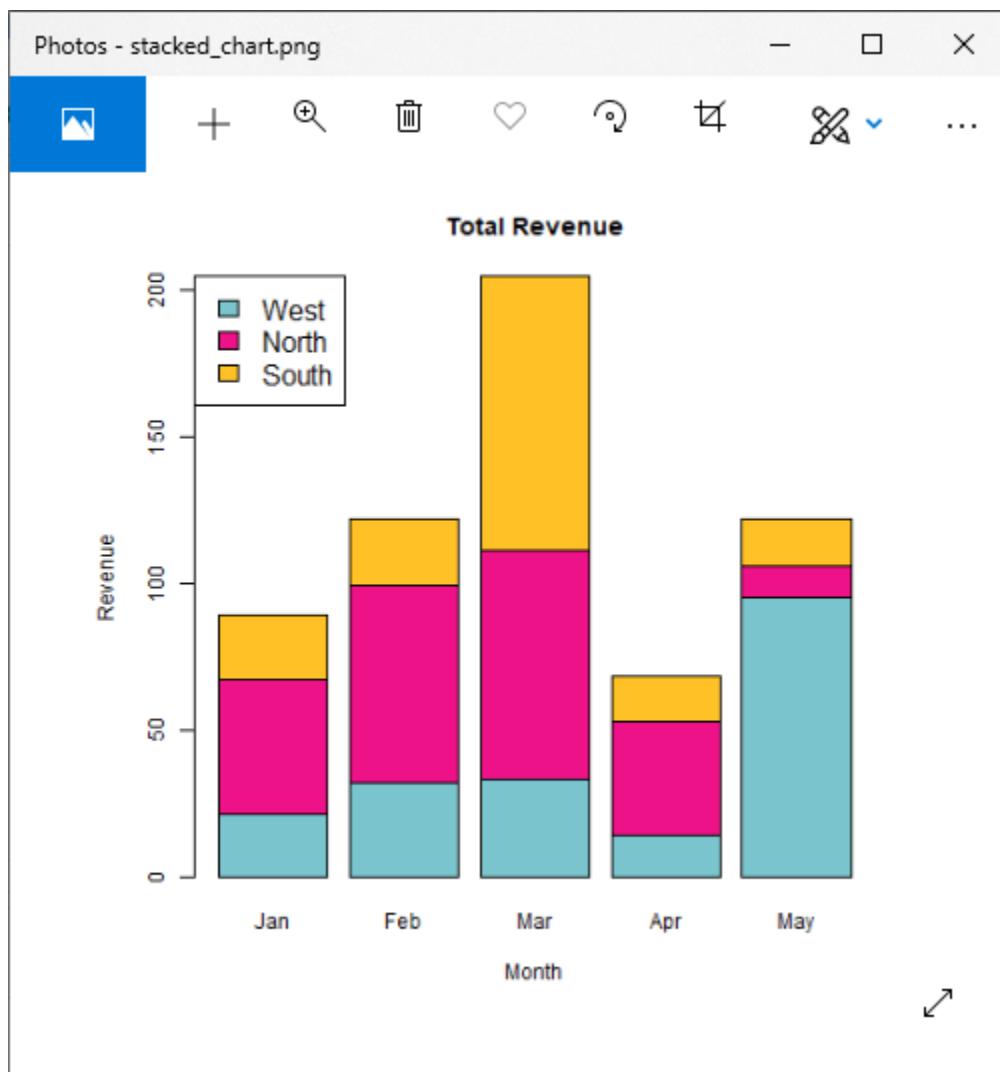
Let's see an example to understand how these charts are created.

Example

```
library(RColorBrewer)  
months <- c("Jan", "Feb", "Mar", "Apr", "May")
```

```
regions <- c("West","North","South")
# Creating the matrix of the values.
Values <- matrix(c(21,32,33,14,95,46,67,78,39,11,22,23,94,15,16), nrow = 3, ncol = 5,
byrow = TRUE)
# Creating the bar chart
barplot(Values, main = "Total Revenue", names.arg = months, xlab = "Month", ylab =
"Revenue", col =c("pink","yellow","brown"))
# Adding the legend to the chart
legend("topleft", regions, cex = 1.3, fill = c("pink","yellow","brown"))
```

Output:



R Boxplot

Boxplots are a measure of how well data is distributed across a data set. This divides the data set into three quartiles. This graph represents the minimum, maximum, average, first quartile, and the third quartile in the data set. Boxplot is also useful in comparing the distribution of data in a data set by drawing a boxplot for each of them.

R provides a `boxplot()` function to create a boxplot. There is the following syntax of `boxplot()` function:

1. `boxplot(x, data, notch, varwidth, names, main)`

Here,

S.No	Parameter	Description
1.	x	It is a vector or a formula.
2.	data	It is the data frame.
3.	notch	It is a logical value set as true to draw a notch.
4.	varwidth	It is also a logical value set as true to draw the width of the box same as the sample size.
5.	names	It is the group of labels that will be printed under each boxplot.
6.	main	It is used to give a title to the graph.

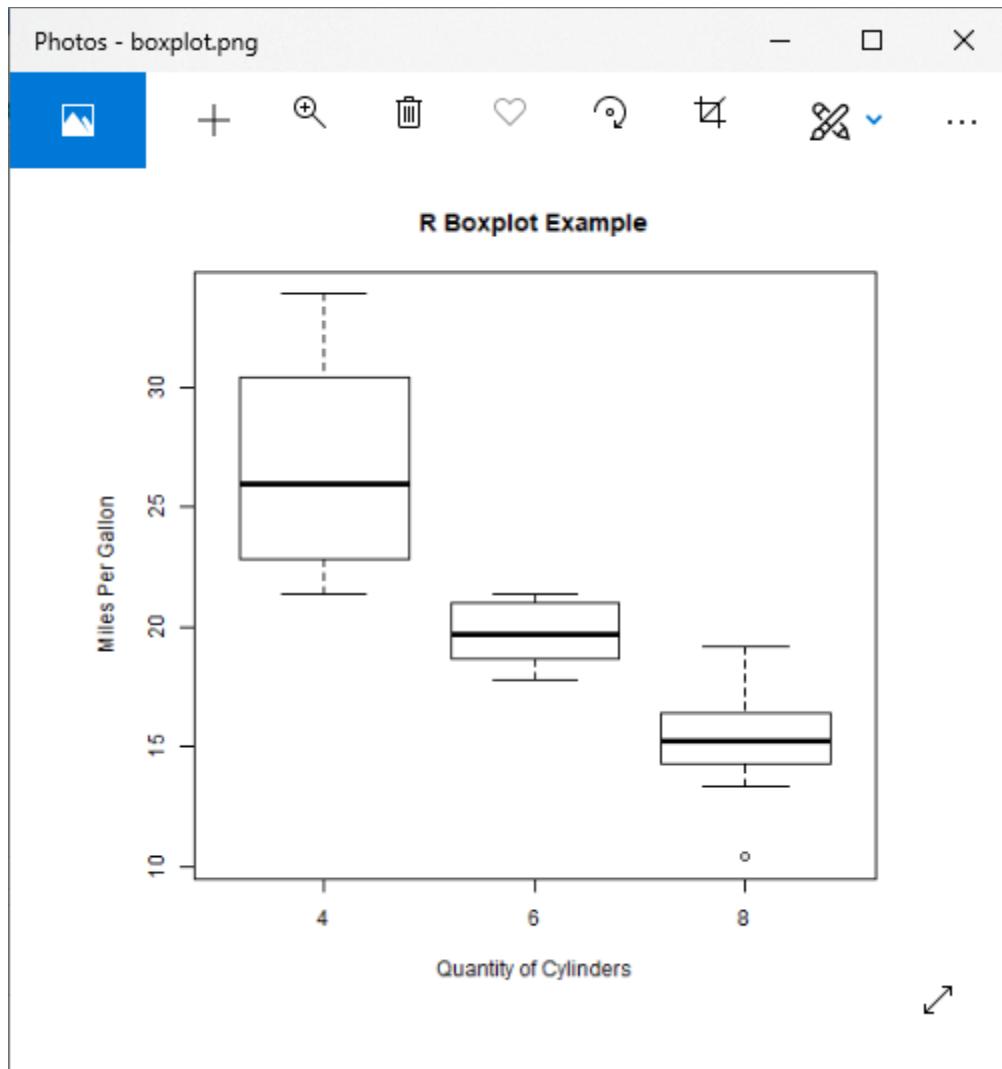
Let's see an example to understand how we can create a boxplot in R. In the below example, we will use the "mtcars" dataset present in the R environment. We will use its two columns only, i.e., "mpg" and "cyl". The below example will create a boxplot graph for the relation between mpg and cyl, i.e., miles per gallon and number of cylinders, respectively.

Example

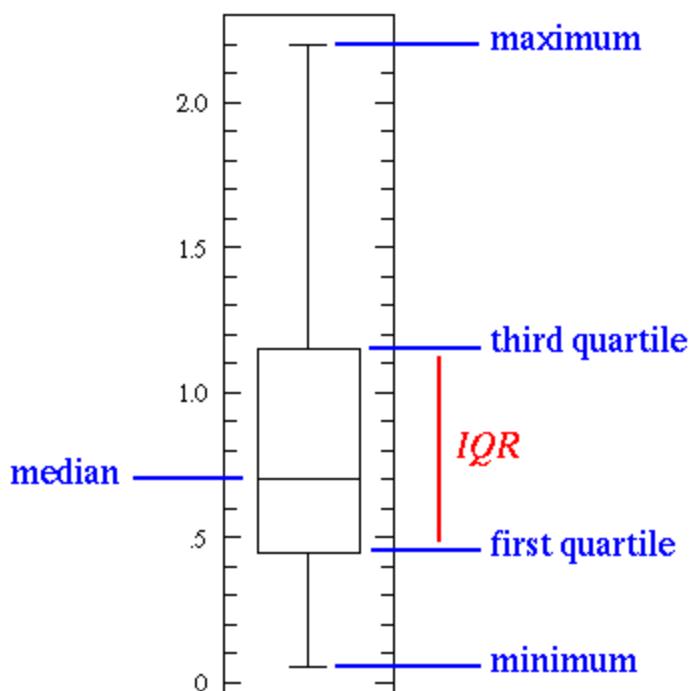
```
# Giving a name to the chart file.
Values <- matrix(c(21,32,33,14,95,46,67,78,39,11,22,23,94,15,16), nrow = 3, ncol = 5, byrow = TRUE)

boxplot(Values, xlab = "Months", ylab = "Marks", main = "Student Result", col="orange")
```

Output:



Quartiles help to give us a fuller picture of our data set as a whole. The first and third quartiles **give us information about the internal structure of our data**. The middle half of the data falls between the first and third quartiles, and is centered about the median.



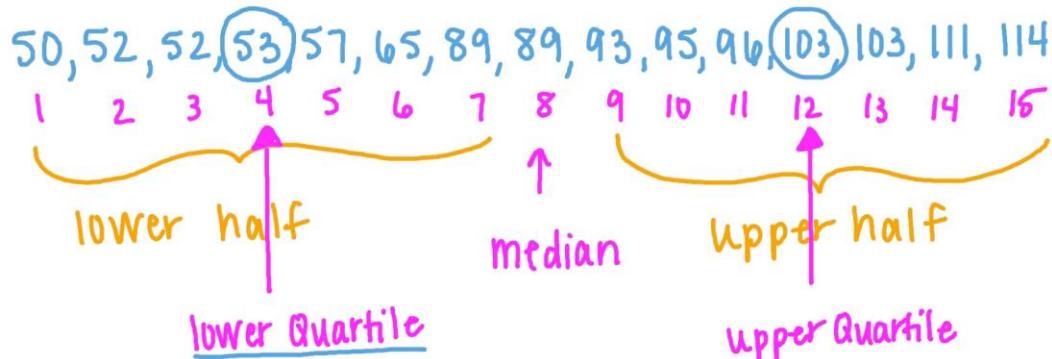
Quartile Formula

$$\text{Lower Quartile (Q1)} = (N+1) \times \frac{1}{4}$$

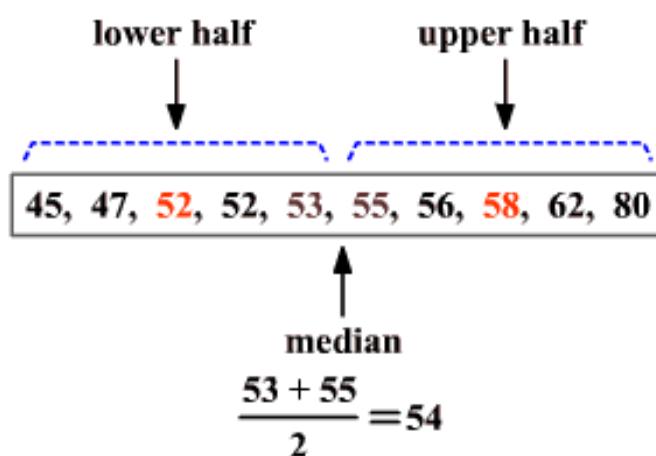
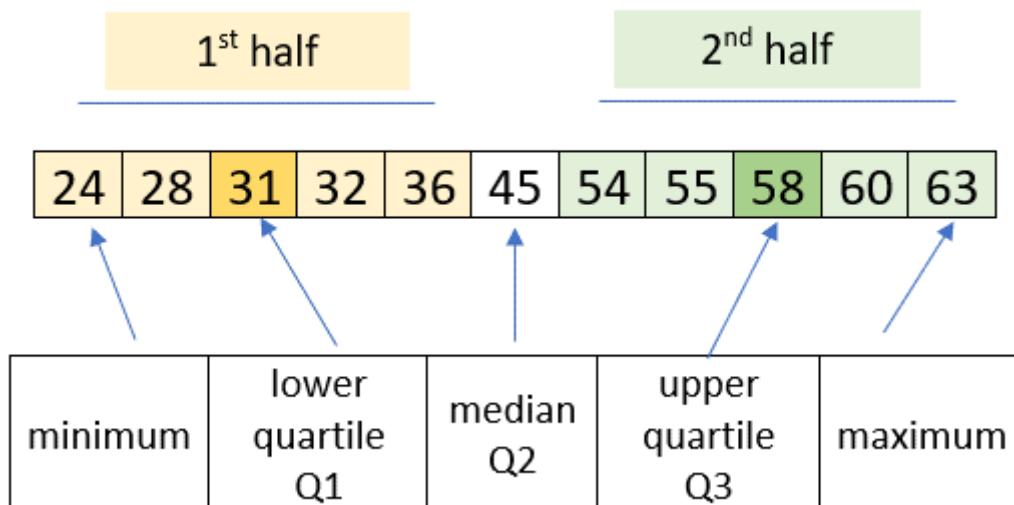
$$\text{Middle Quartile (Q2)} = (N+1) \times \frac{2}{4}$$

$$\text{Upper Quartile (Q3)} = (N+1) \times \frac{3}{4}$$

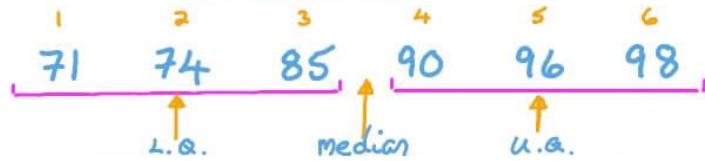
Determine the upper and lower quartiles of the following set of data:
 114, 103, 50, 52, 95, 103, 93, 53, 65, 57, 52, 89, 111, 89 and 96.



Lower Quartile is 53 and Upper Quartile is 103



David's history test scores are 74, 96, 85, 90, 71, and 98. Determine the upper and lower quartiles of his scores.



$$\text{Median - middle value} \quad \frac{85+90}{2} = 87.5 \quad \checkmark \quad \text{Median position : } \frac{n+1}{2} \quad \frac{6+1}{2} = 3.5$$

Lower quartile - center of bottom half

Upper quartile - center of top half

74
96

$$\text{L.Q. position : } \frac{n+1}{4} \quad \frac{6+1}{4} = 1.75 \quad \text{Q}^{\text{nd}}$$

$$\text{U.Q. position : } \frac{3(n+1)}{4} \quad \frac{3(6+1)}{4} = 5.25 \quad \text{Q}^{\text{th}}$$

Boxplot using notch

In R, we can draw a boxplot using a notch. It helps us to find out how the medians of different data groups match with each other. Let's see an example to understand how a boxplot graph is created using notch for each of the groups.

In our below example, we will use the same dataset ?mtcars."

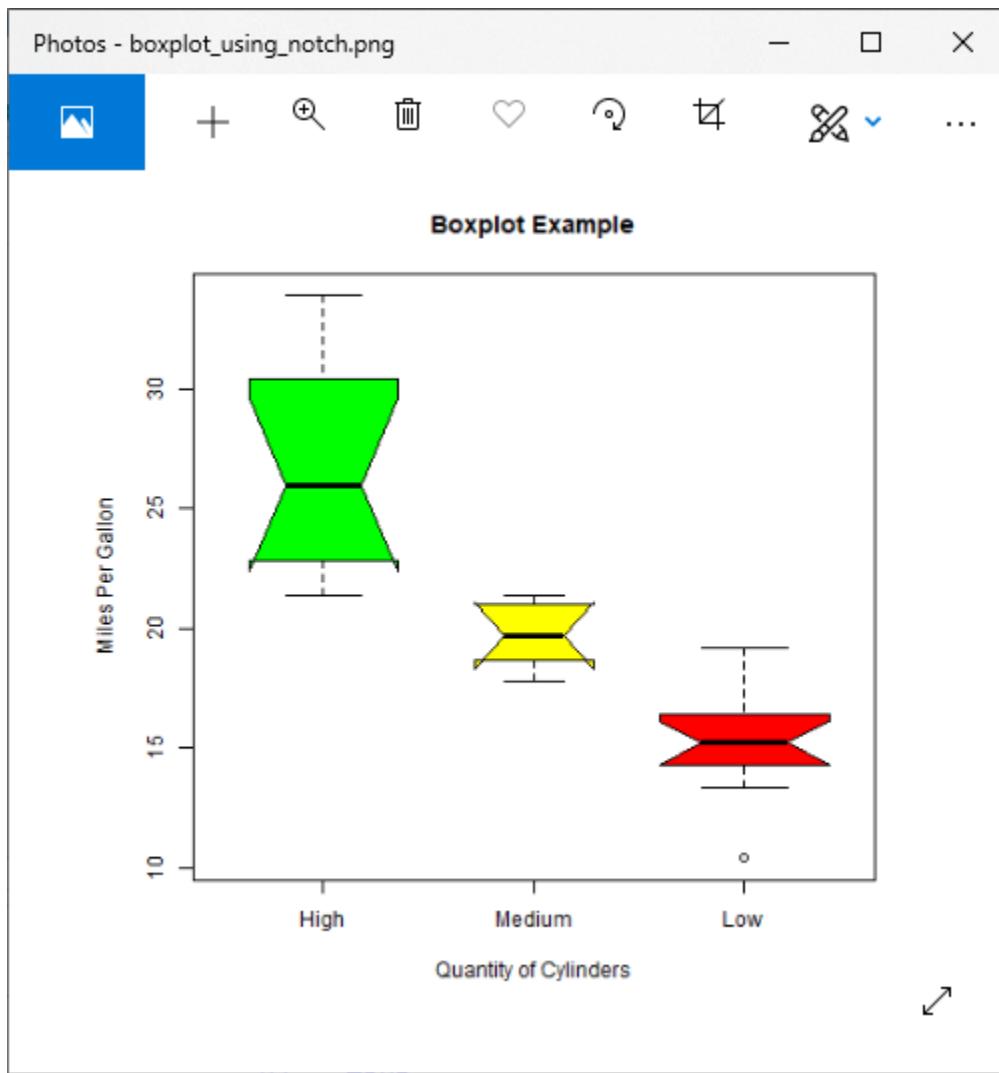
Example

Giving a name to our chart.

```
Values <- matrix(c(21,32,33,14,95,46,67,78,39,11,22,23,94,15,16), nrow = 3, ncol = 5, byrow = TRUE)
```

```
boxplot(Values, xlab = "Months", ylab = "Marks", main = "Student Result",notch = TRUE,col="orange")
```

Output:



Violin Plots

R provides an additional plotting scheme which is created with the combination of a **boxplot** and a **kernel density** plot. The violin plots are created with the help of `vioplot()` function present in the `vioplot` package.

Let's see an example to understand the creation of the violin plot.

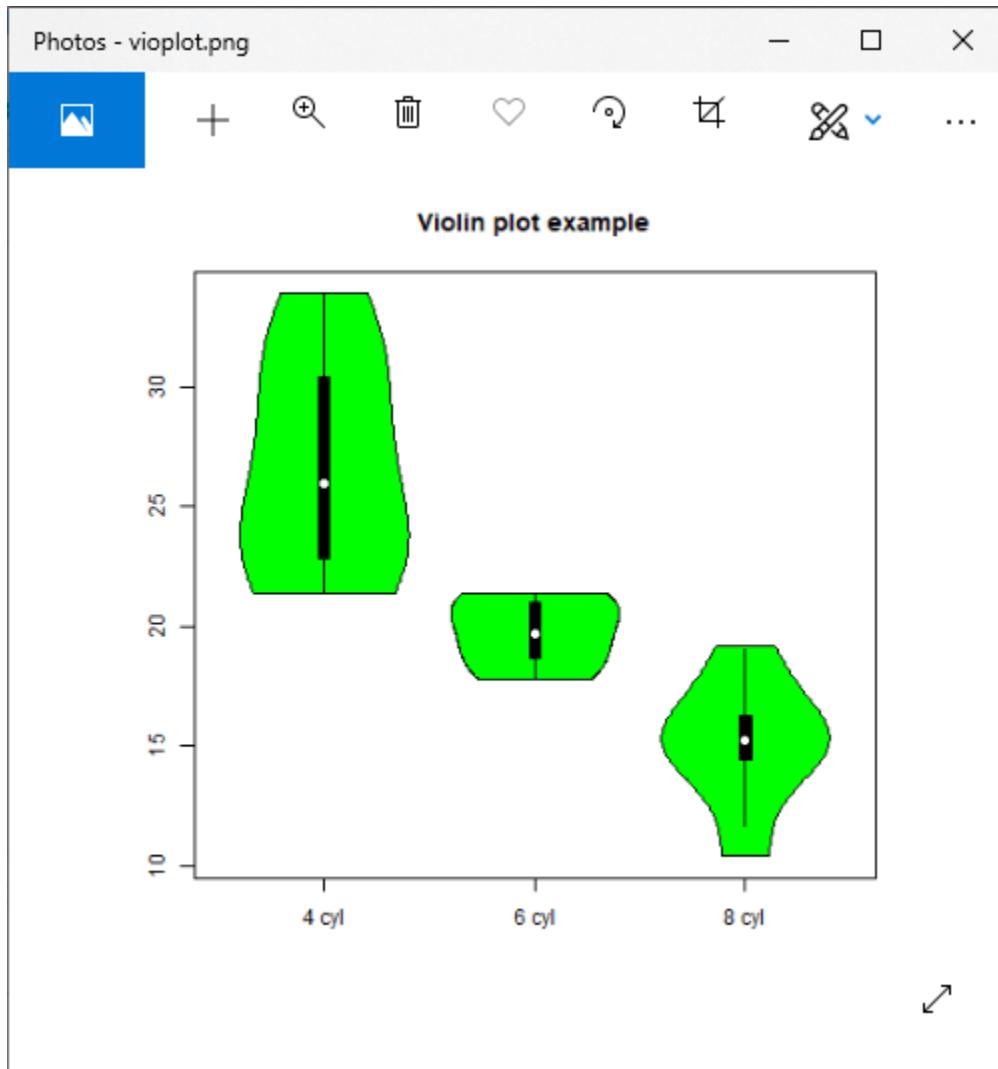
Example

Loading the `vioplot` package

```
library(vioplot)
#Creating data for vioplot function
x1 <- mtcars$mpg[mtcars$cyl==4]
x2 <- mtcars$mpg[mtcars$cyl==6]
x3 <- mtcars$mpg[mtcars$cyl==8]
```

```
#Creating vioplot function
vioplot(x1, x2, x3, names=c("4 cyl", "6 cyl", "8 cyl"), col="green")
#Setting title
title("Violin plot example")
```

Output:



Bagplot- 2-Dimensional Boxplot Extension

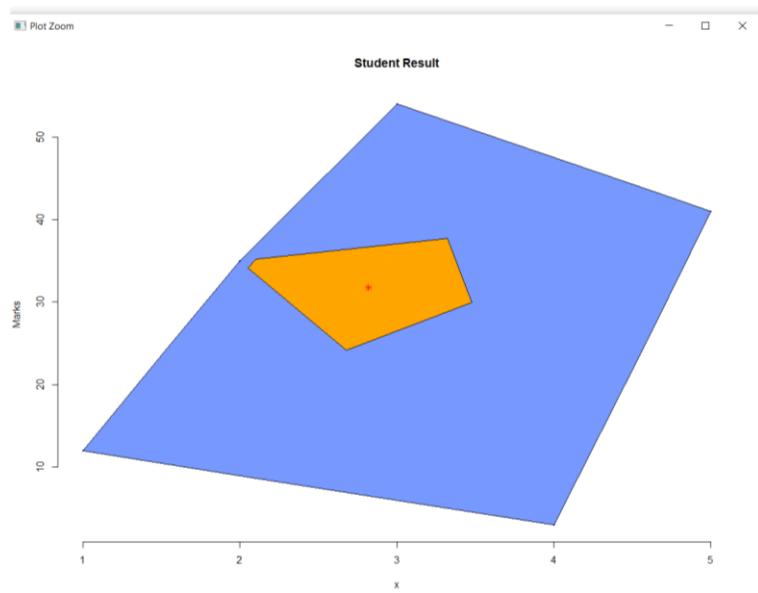
The bagplot(x, y) function in the **aplypack** package provides a biennial version of the univariate boxplot. The bag contains 50% of all points. The bivariate median is approximate. The fence separates itself from the outside points, and the outlays are displayed.

Let's see an example to understand how we can create a two-dimensional boxplot extension in R.

Example

```
# Loading aplpack package  
  
library(aplpack)  
  
H <- c(12,35,54,3,41)  
  
M<- c("Feb","Mar","Apr","May","Jun")  
  
bagplot(H, M = "Months", ylab = "Marks", main = "Student Result")
```

Output:



R Histogram

A histogram is a type of bar chart which shows the frequency of the number of values which are compared with a set of values ranges. The histogram is used for the distribution, whereas a bar chart is used for comparing different entities. In the histogram, each bar represents the height of the number of values present in the given range.

For creating a histogram, R provides `hist()` function, which takes a vector as an input and uses more parameters to add more functionality. There is the following syntax of `hist()` function:

1. `hist(v,main,xlab,ylab,xlim,ylim,breaks,col,border)`

Here,

S.No	Parameter	Description
1.	v	It is a vector that contains numeric values.
2.	main	It indicates the title of the chart.
3.	col	It is used to set the color of the bars.
4.	border	It is used to set the border color of each bar.
5.	xlab	It is used to describe the x-axis.
6.	ylab	It is used to describe the y-axis.
7.	xlim	It is used to specify the range of values on the x-axis.
8.	ylim	It is used to specify the range of values on the y-axis.
9.	breaks	It is used to mention the width of each bar.

Let's see an example in which we create a simple histogram with the help of required parameters like v, main, col, etc.

Example

```
# Creating data for the graph.
v <- c(12,24,16,38,21,13,55,17,39,10,60)
```

```
# Creating the histogram.
hist(v,xlab = "Weight",ylab="Frequency",col = "green",border = "red")
```

Output:

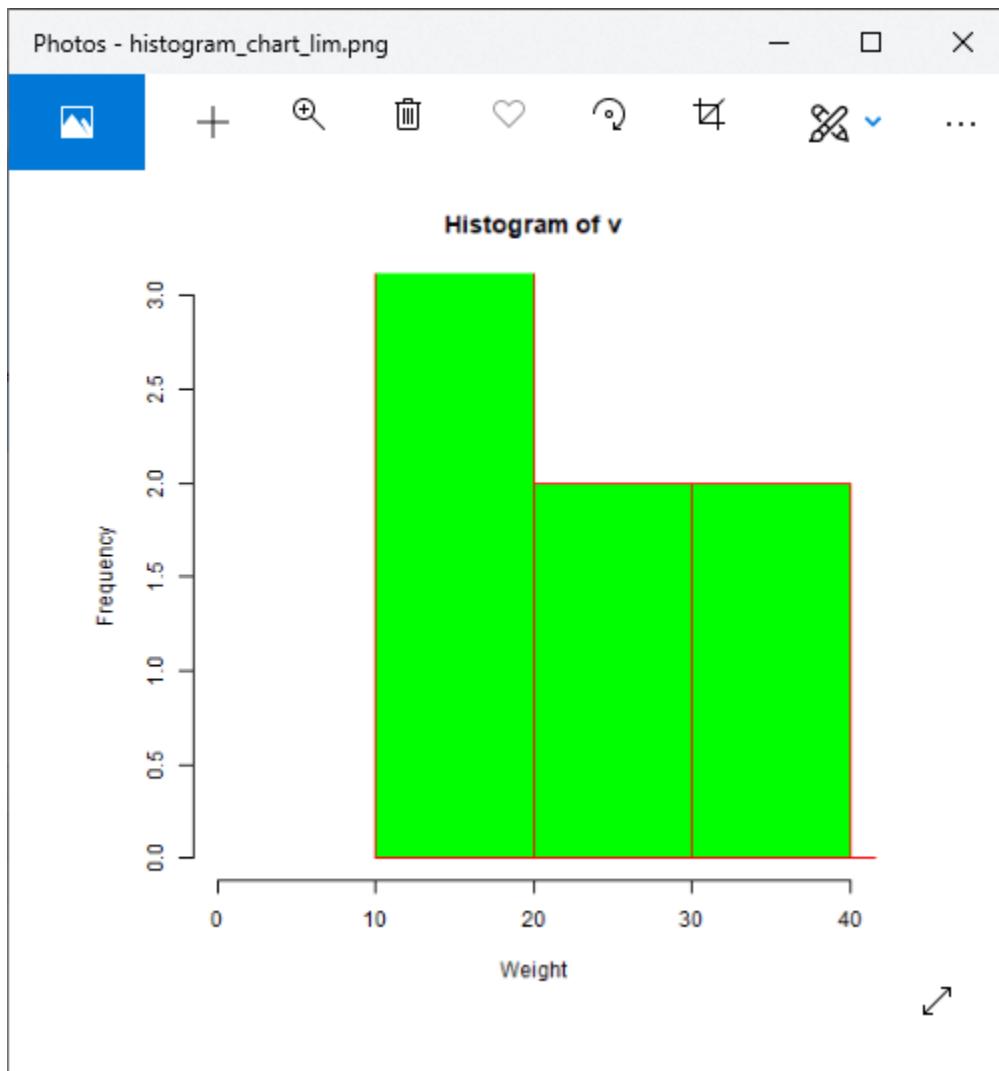


Let's see some more examples in which we have used different parameters of `hist()` function to add more functionality or to create a more attractive chart.

Example: Use of `xlim` & `ylim` parameter

```
1. # Creating data for the graph.  
v <- c(12,24,16,38,21,13,55,17,39,10,60)  
# Creating the histogram.  
hist(v,xlab = "Weight",ylab="Frequency",col = "green",border = "red",xlim = c(0,40), ylim = c(0,3), breaks = 5)
```

Output:



Example: Finding return value of hist()

1. # Creating data for the graph.
2. v <- c(12,24,16,38,21,13,55,17,39,10,60)
- 3.
4. # Giving a name to the chart file.
5. png(file = "histogram_chart_lim.png")
6. # Creating the histogram.
7. m<-hist(v)
8. m

Output:

```
Command Prompt
Microsoft Windows [Version 10.0.18362.239]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\ajeet>cd R

C:\Users\ajeet\R>Rscript histo.R
$breaks
[1] 10 20 30 40 50 60

$counts
[1] 5 2 2 0 2

$density
[1] 0.04545455 0.01818182 0.01818182 0.00000000 0.01818182

$mid
[1] 15 25 35 45 55

$xname
[1] "v"

$equidist
[1] TRUE

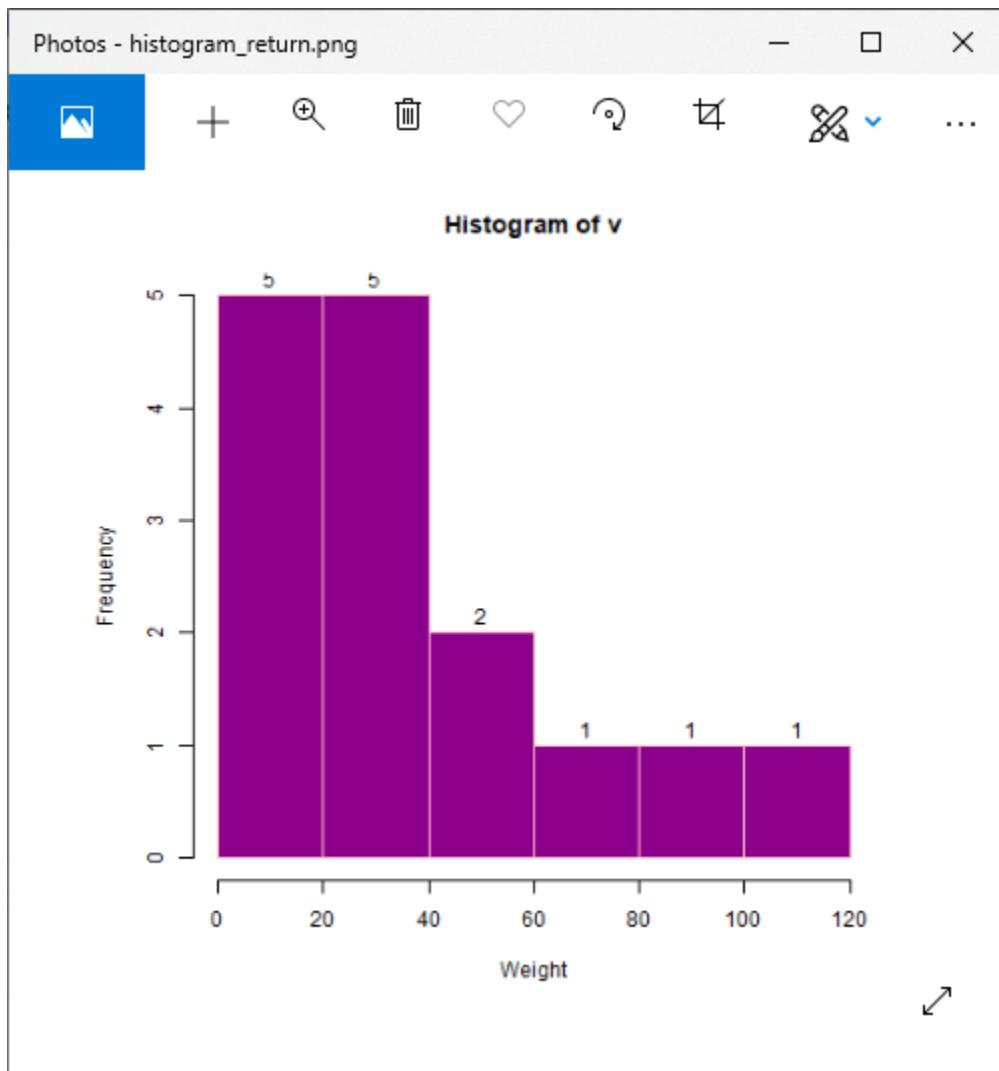
attr(,"class")
[1] "histogram"
C:\Users\ajeet\R>
```

Example: Using histogram return values for labels using `text()`

```
# Creating data for the graph.
v <- c(12,24,16,38,21,13,55,17,39,10,60,120,40,70,90)

# Creating the histogram.
m<-
  hist(v,xlab = "Weight",ylab="Frequency",col = "darkmagenta",border = "pink", br
eaks = 5)
#Setting labels
text(m$mids,m$counts,labels=m$counts, adj=c(0.5, -0.5))
```

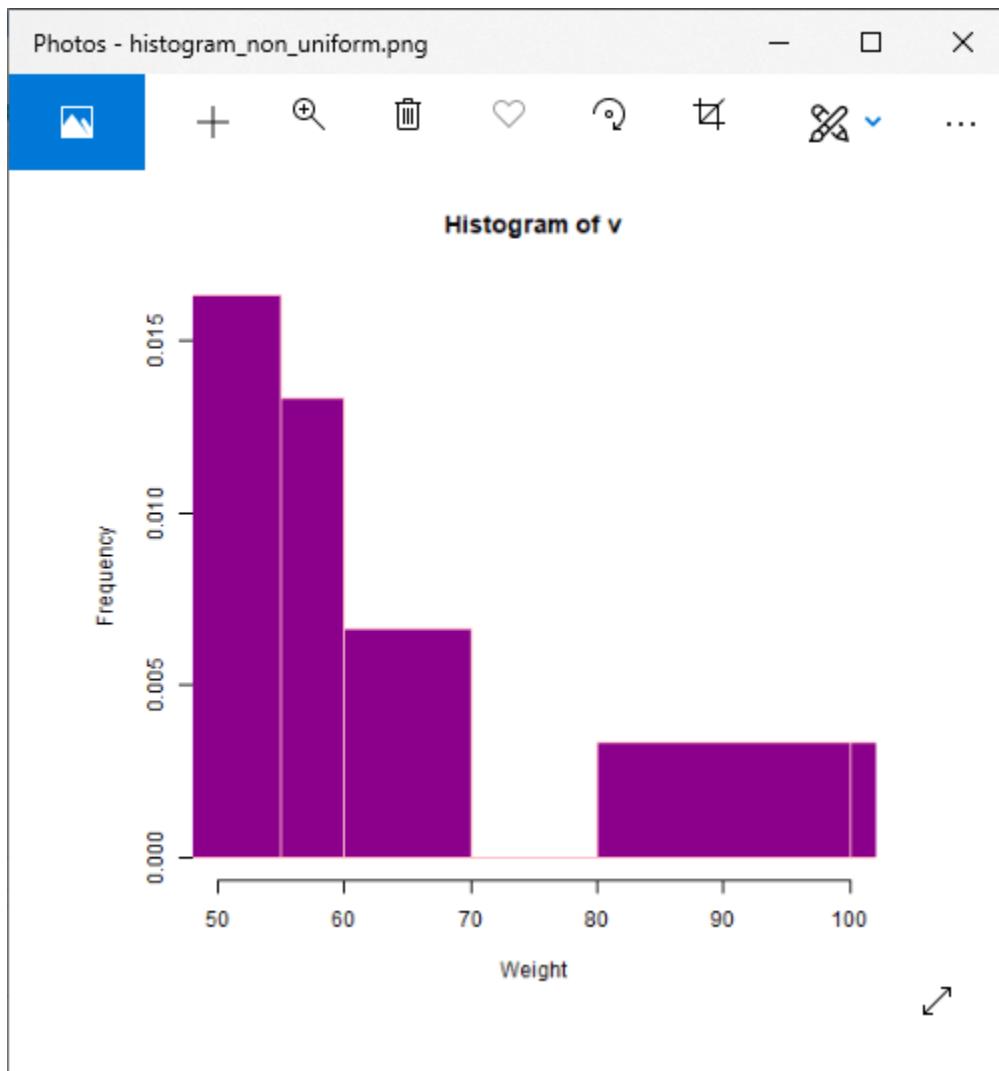
Output:



Example: Histogram using non-uniform width

1. # Creating data for the graph.
2. v <- c(12,24,16,38,21,13,55,17,39,10,60,120,40,70,90)
3. # Giving a name to the chart file.
4. png(file = "histogram_non_uniform.png")
5. # Creating the histogram.
6. hist(v,xlab = "Weight",ylab="Frequency",xlim=c(50,100),col = "darkmagenta",border = "pink", breaks=c(10,55,60,70,75,80,100,120))
7. # Saving the file.
8. dev.off()

Output:



R Line Graphs

A line graph is a pictorial representation of information which changes continuously over time. A line graph can also be referred to as a line chart. Within a line graph, there are points connecting the data to show the continuous change. The lines in a line graph can move up and down based on the data. We can use a line graph to compare different events, information, and situations.

A line chart is used to connect a series of points by drawing line segments between them. Line charts are used in identifying the trends in data. For line graph construction, R provides `plot()` function, which has the following syntax:

1. `plot(v,type,col,xlab,ylab)`

Here,

S.No	Parameter	Description
1.	v	It is a vector which contains the numeric values.
2.	type	This parameter takes the value ?l? to draw only the lines or ?p? to draw only the points and "o" to draw both lines and points.
3.	xlab	It is the label for the x-axis.
4.	ylab	It is the label for the y-axis.
5.	main	It is the title of the chart.
6.	col	It is used to give the color for both the points and lines

Let's see a basic example to understand how plot() function is used to create the line graph:

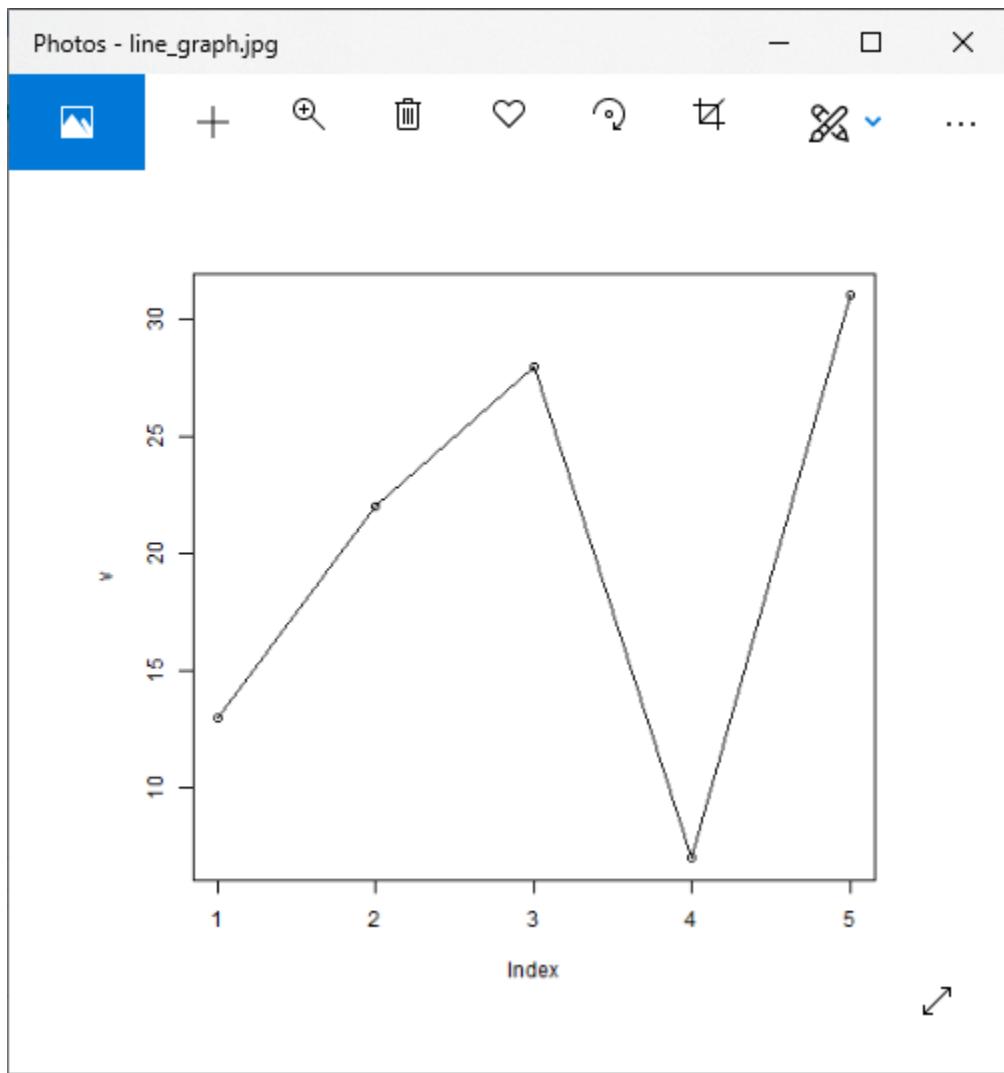
Example

1. # Creating the data for the chart.

```
v <- c(13,22,28,7,31)
```

```
plot(v,type = "o")
```

Output:



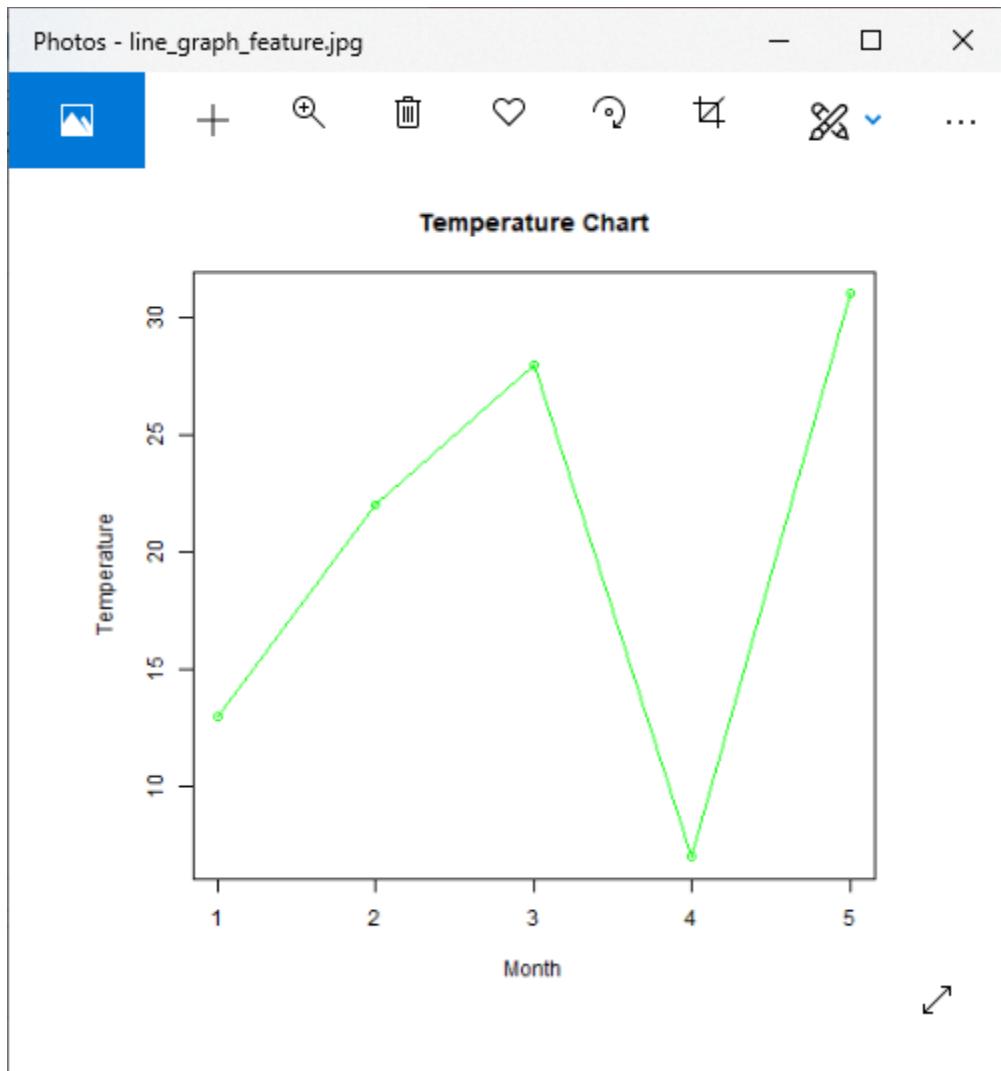
Line Chart Title, Color, and Labels

Like other graphs and charts, in line chart, we can add more features by adding more parameters. We can add the colors to the lines and points, add labels to the axis, and can give a title to the chart. Let's see an example to understand how these parameters are used in `plot()` function to create an attractive line graph.

Example

1. # Creating the data for the chart.
2. `v <- c(13,22,28,7,31)`
3. # Giving a name to the chart file.
4. `png(file = "line_graph_feature.jpg")`
5. # Plotting the bar chart.
6. `plot(v,type = "o",col="green",xlab="Month",ylab="Temperature")`
7. # Saving the file.
8. `dev.off()`

Output:



Line Charts Containing Multiple Lines

In our previous examples, we created line graphs containing only one line in each graph. R allows us to create a line graph containing multiple lines. R provides `lines()` function to create a line in the line graph.

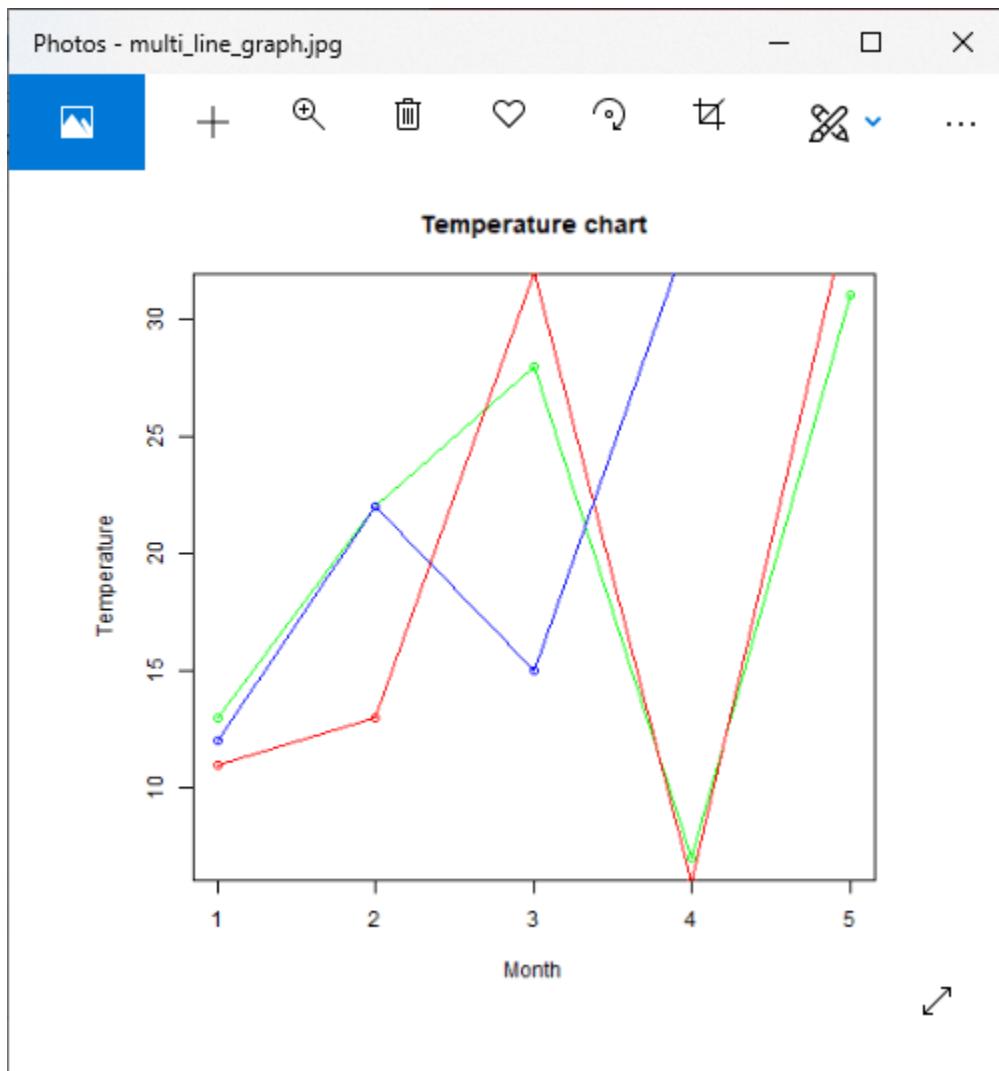
The `lines()` function takes an additional input vector for creating a line. Let's see an example to understand how this function is used:

Example

```
# Creating the data for the chart.  
v <- c(13,22,28,7,31)  
w <- c(11,13,32,6,35)  
x <- c(12,22,15,34,35)
```

```
plot(v,type = "o",col="green",xlab="Month",ylab="Temperature")
lines(w, type = "o", col = "red")
lines(x, type = "o", col = "blue")
```

Output:



Line Graph using ggplot2

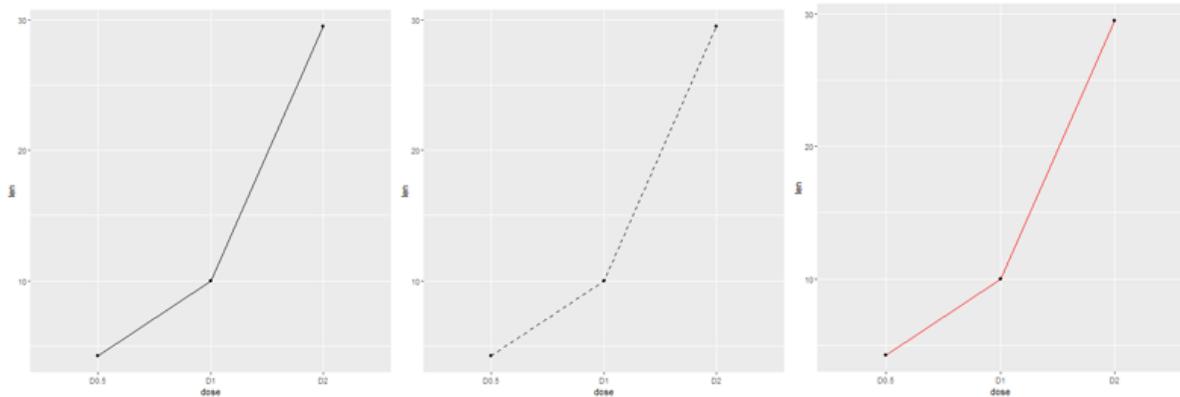
In R, there is another way to create a line graph i.e. the use of ggplot2 packages. The ggplot2 package provides `geom_line()`, `geom_step()` and `geom_path()` function to create line graph. To use these functions, we first have to install the ggplot2 package and then we load it into the current working library.

Let's see an example to understand how ggplot2 is used to create a line graph. In the below example, we will use the predefined ToothGrowth dataset, which describes the effect of vitamin C on tooth growth in Guinea pigs.

Example

```
library(ggplot2)
#Creating data for the graph
data_frame<- data.frame(dose=c("D0.5", "D1", "D2"),
len=c(4.2, 10, 29.5))
# Basic line plot with points
ggplot(data=data_frame, aes(x=dose, y=len, group=1)) +geom_line()+geom_point()
# Change the line type
ggplot(data=df, aes(x=dose, y=len, group=1)) +geom_line(linetype = "dashed")+geom_p
oint()
# Change the color
ggplot(data=df, aes(x=dose, y=len, group=1)) +geom_line(color="red")+geom_point()
```

Output:



R Scatterplots

The scatter plots are used to compare variables. A comparison between variables is required when we need to define how much one variable is affected by another variable. In a scatterplot, the data is represented as a collection of points. Each point on the scatterplot defines the values of the two variables. One variable is selected for the vertical axis and other for the horizontal axis. In R, there are two ways of creating scatterplot, i.e., using `plot()` function and using the `ggplot2` package's functions.

There is the following syntax for creating scatterplot in R:

1. `plot(x, y, main, xlab, ylab, xlim, ylim, axes)`

Here,

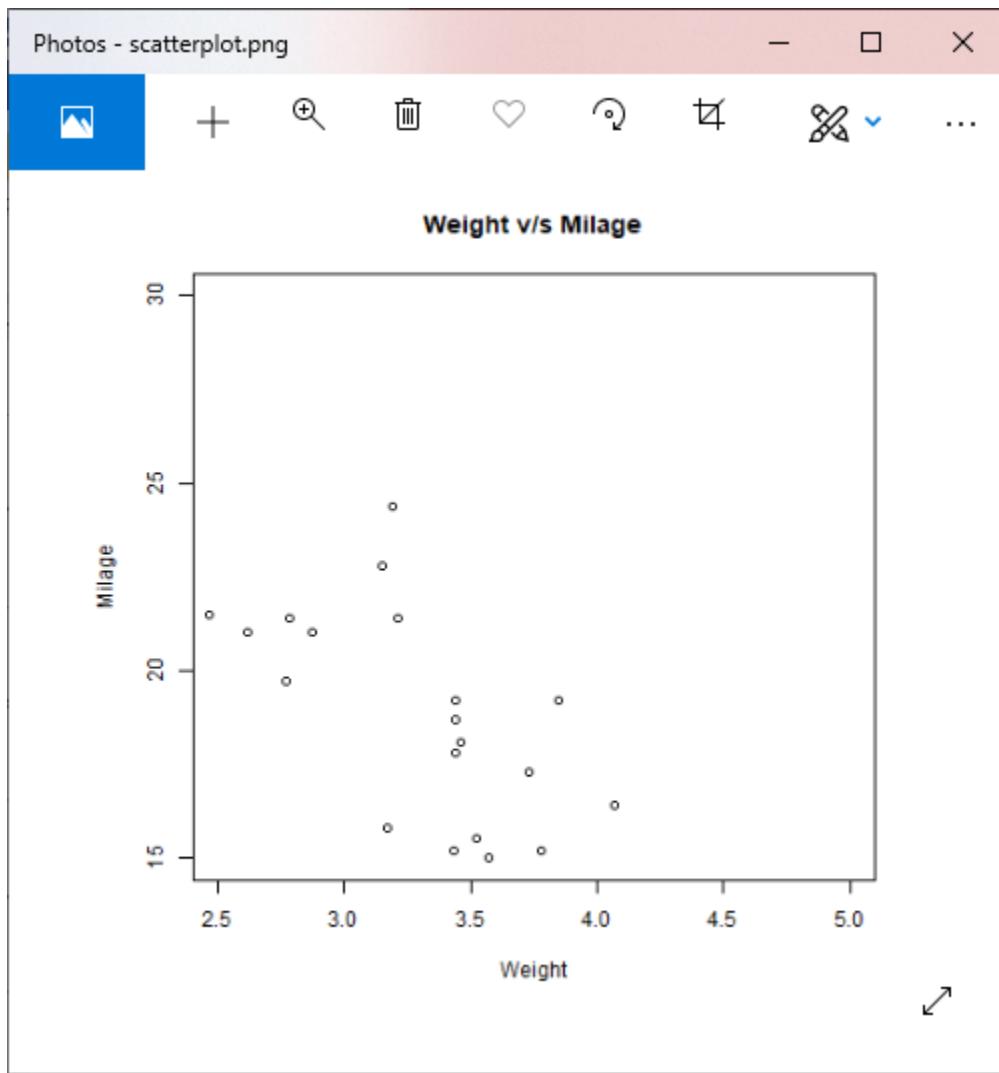
S.No	Parameters	Description
1.	x	It is the dataset whose values are the horizontal coordinates.
2.	y	It is the dataset whose values are the vertical coordinates.
3.	main	It is the title of the graph.
4.	xlab	It is the label on the horizontal axis.
5.	ylab	It is the label on the vertical axis.
6.	xlim	It is the limits of the x values which is used for plotting.
7.	ylim	It is the limits of the values of y, which is used for plotting.
8.	axes	It indicates whether both axes should be drawn on the plot.

Let's see an example to understand how we can construct a scatterplot using the `plot` function. In our example, we will use the dataset "mtcars", which is the predefined dataset available in the R environment.

Example

```
1. #Fetching two columns from mtcars  
data <-mtcars[,c('wt','mpg')]  
# Plotting the chart for cars with weight between 2.5 to 5 and mileage between 1  
5 and 30.  
plot(x = data$wt,y = data$mpg, xlab = "Weight", ylab = "Milage", xlim = c(2.5,5), ylim = c  
(15,30), main = "Weight v/s Milage")
```

Output:



Scatterplot using ggplot2

In R, there is another way for creating scatterplot i.e. with the help of ggplot2 package.

The ggplot2 package provides `ggplot()` and `geom_point()` function for creating a scatterplot. The `ggplot()` function takes a series of the input item. The first parameter is an input vector, and the second is the `aes()` function in which we add the x-axis and y-axis.

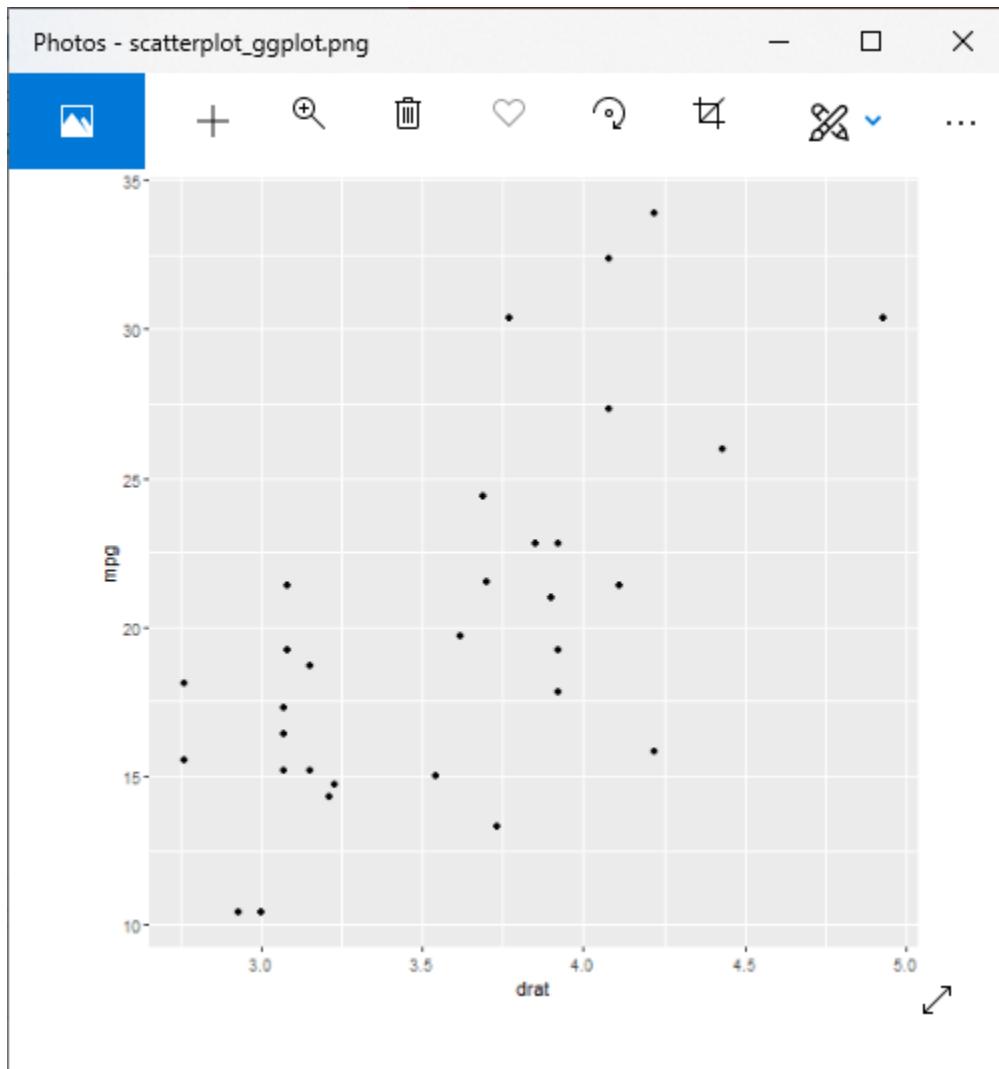
Let's start understanding how the ggplot2 package is used with the help of an example where we have used the familiar dataset "mtcars".

Example

1. #Loading ggplot2 package
2. `library(ggplot2)`
3. # Giving a name to the chart file.

```
4. png(file = "scatterplot_ggplot.png")
5. # Plotting the chart using ggplot() and geom_point() functions.
6. ggplot(mtcars, aes(x = drat, y = mpg)) +geom_point()
7. # Saving the file.
8. dev.off()
```

Output:



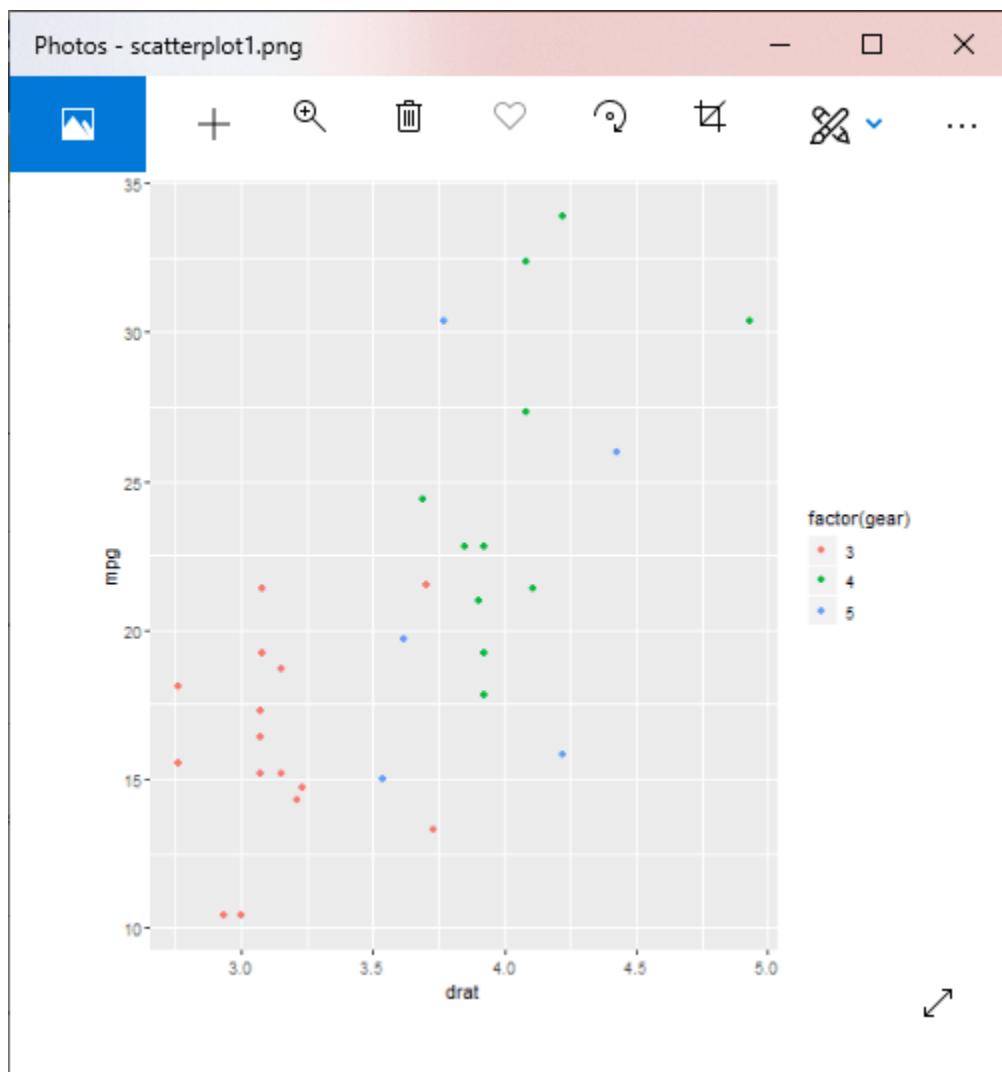
We can add more features and make a more attractive scatter plots also. Below are some examples in which different parameters are added.

Example 1: Scatterplot with groups

1. #Loading ggplot2 package
2. library(ggplot2)
3. # Giving a name to the chart file.
4. png(file = "scatterplot1.png")

```
5. # Plotting the chart using ggplot() and geom_point() functions.  
6. #The aes() function inside the geom_point() function controls the color of the group.  
7. ggplot(mtcars, aes(x = drat, y = mpg)) +  
8. geom_point(aes(color=factor(gear)))  
9. # Saving the file.  
10. dev.off()
```

Output:

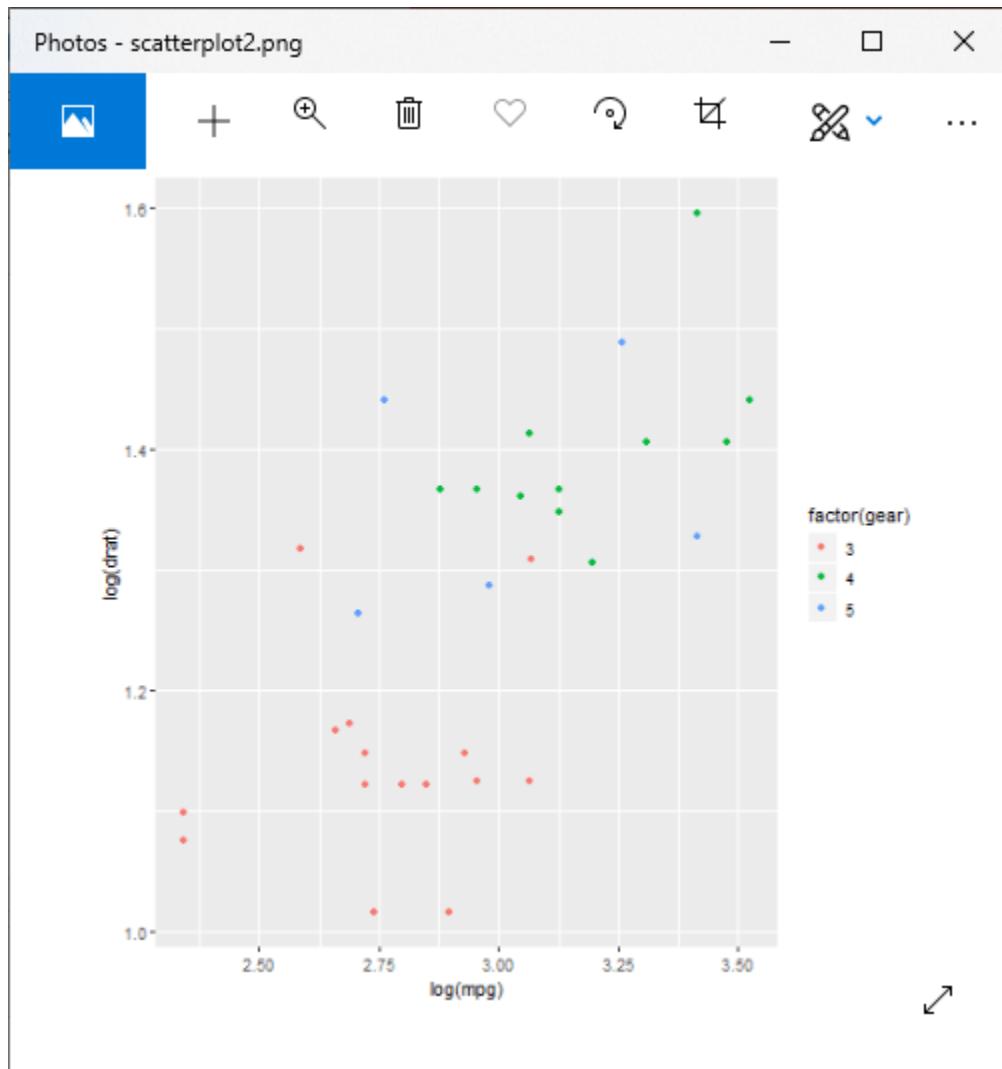


Example 2: Changes in axis

1. #Loading ggplot2 package
 2. library(ggplot2)
 3. # Giving a name to the chart file.
 4. png(file = "scatterplot2.png")
 5. # Plotting the chart using ggplot() and geom_point() functions.

6. #The aes() function inside the geom_point() function controls the color of the group.
7. ggplot(mtcars, aes(x = log(mpg), y = log(drat))) +geom_point(aes(color=factor(gear)))
8. # Saving the file.
9. dev.off()

Output:

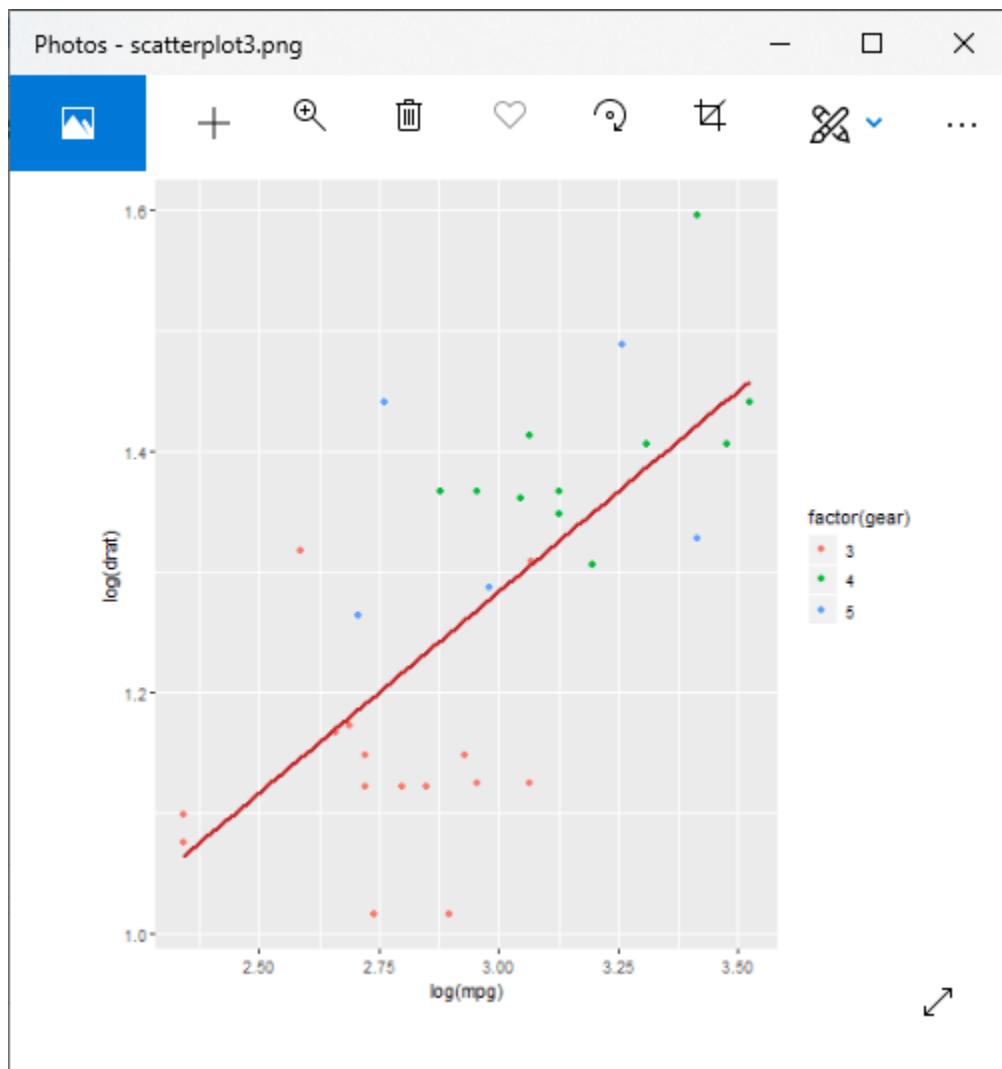


Example 3: Scatterplot with fitted values

1. #Loading ggplot2 package
2. library(ggplot2)
3. # Giving a name to the chart file.
4. png(file = "scatterplot3.png")
5. #Creating scatterplot with fitted values.
6. # An additional function stat_smooth is used for linear regression.

7. `ggplot(mtcars, aes(x = log(mpg), y = log(drat))) +geom_point(aes(color = factor(gear))) + stat_smooth(method = "lm", col = "#C42126", se = FALSE, size = 1)`
8. #in above example lm is used for linear regression and se stands for standard error.
9. # Saving the file.
10. `dev.off()`

Output:



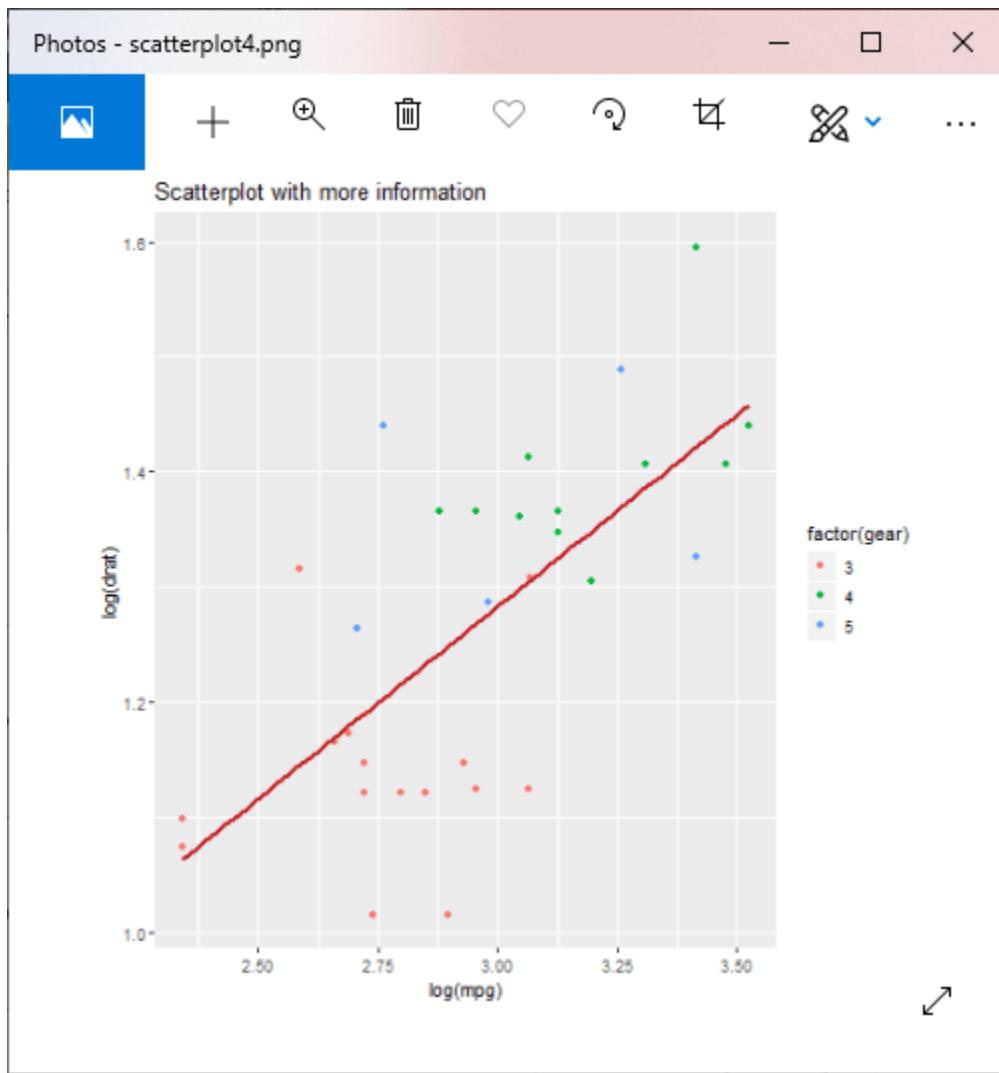
Adding information to the graph

Example 4: Adding title

1. #Loading ggplot2 package
2. `library(ggplot2)`
3. # Giving a name to the chart file.
4. `png(file = "scatterplot4.png")`

```
5. #Creating scatterplot with fitted values.  
6. # An additional function stat_smooth is used for linear regression.  
7. new_graph<-  
  ggplot(mtcars, aes(x = log(mpg), y = log(drat))) +geom_point(aes(color = fact  
  or(gear))) +  
8. stat_smooth(method = "lm",col = "#C42126",se = FALSE,size = 1)  
9. #in above example lm is used for linear regression and se stands for standard  
  error.  
10. new_graph+  
11. labs(  
12.   title = "Scatterplot with more information"  
13.)  
14. # Saving the file.  
15. dev.off()
```

Output:



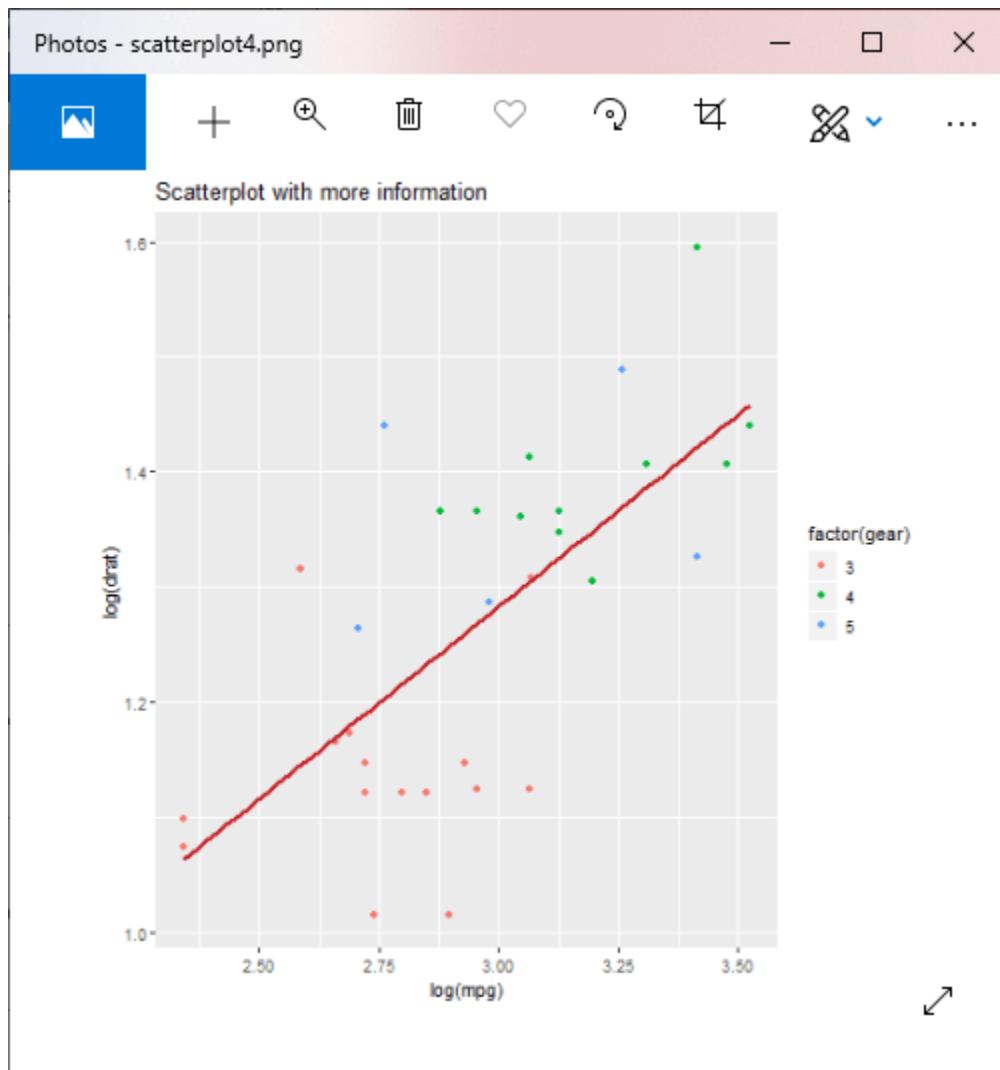
Example 5: Adding title with dynamic name

1. #Loading ggplot2 package
2. library(ggplot2)
3. # Giving a name to the chart file.
4. png(file = "scatterplot5.png")
5. #Creating scatterplot with fitted values.
6. # An additional function stat_smooth is used for linear regression.
7. new_graph<-


```
ggplot(mtcars, aes(x = log(mpg), y = log(drat))) +geom_point(aes(color = factor(gear))) +
stat_smooth(method = "lm", col = "#C42126", se = FALSE, size = 1)
```
8. #in above example lm is used for linear regression and se stands for standard error.
9. #Finding mean of mpg
10. mean_mpg<- mean(mtcars\$mpg)

```
12. #Adding title with dynamic name  
13. new_graph + labs(  
14.     title = paste("Adding additiona information. Average mpg is", mean_mpg)  
15.)  
16. # Saving the file.  
17. dev.off()
```

Output:

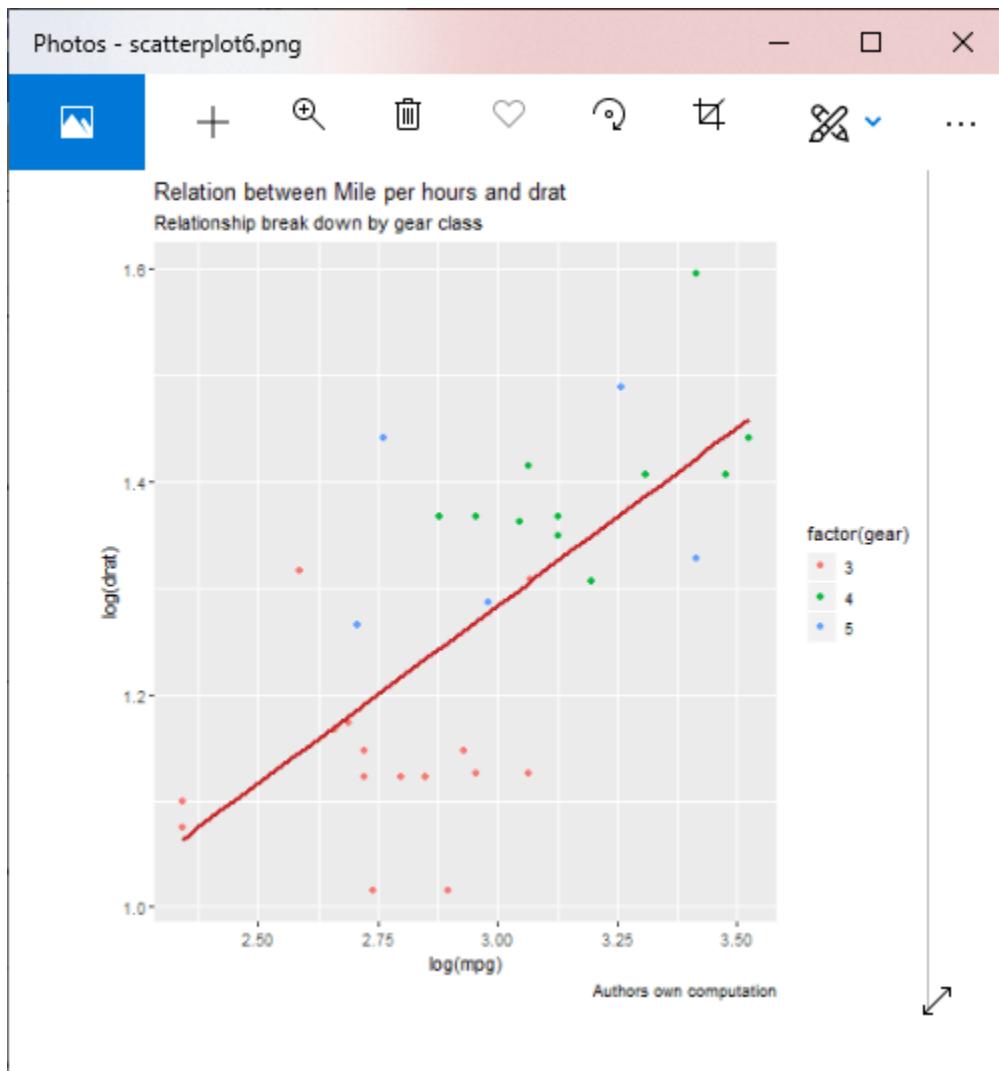


Example 6: Adding a sub-title

1. #Loading ggplot2 package
2. library(ggplot2)
3. # Giving a name to the chart file.
4. png(file = "scatterplot6.png")
5. #Creating scatterplot with fitted values.

```
6. # An additional function stat_smooth is used for linear regression.  
7. new_graph<-  
  ggplot(mtcars, aes(x = log(mpg), y = log(drat))) +geom_point(aes(color = fact  
  or(gear))) +  
8. stat_smooth(method = "lm",col = "#C42126",se = FALSE,size = 1)  
9. #in above example lm is used for linear regression and se stands for standard  
  error.  
10. #Adding title with dynamic name  
11. new_graph + labs(  
12.   title =  
13.     "Relation between Mile per hours and drat",  
14.   subtitle =  
15.     "Relationship break down by gear class",  
16.   caption = "Authors own computation"  
17.)  
18. # Saving the file.  
19. dev.off()
```

Output:



Example 7: Changing name of x-axis and y-axis

1. #Loading ggplot2 package
2. library(ggplot2)
3. # Giving a name to the chart file.
4. png(file = "scatterplot7.png")
5. #Creating scatterplot with fitted values.
6. # An additional function stat_smooth is used for linear regression.
7. new_graph<-

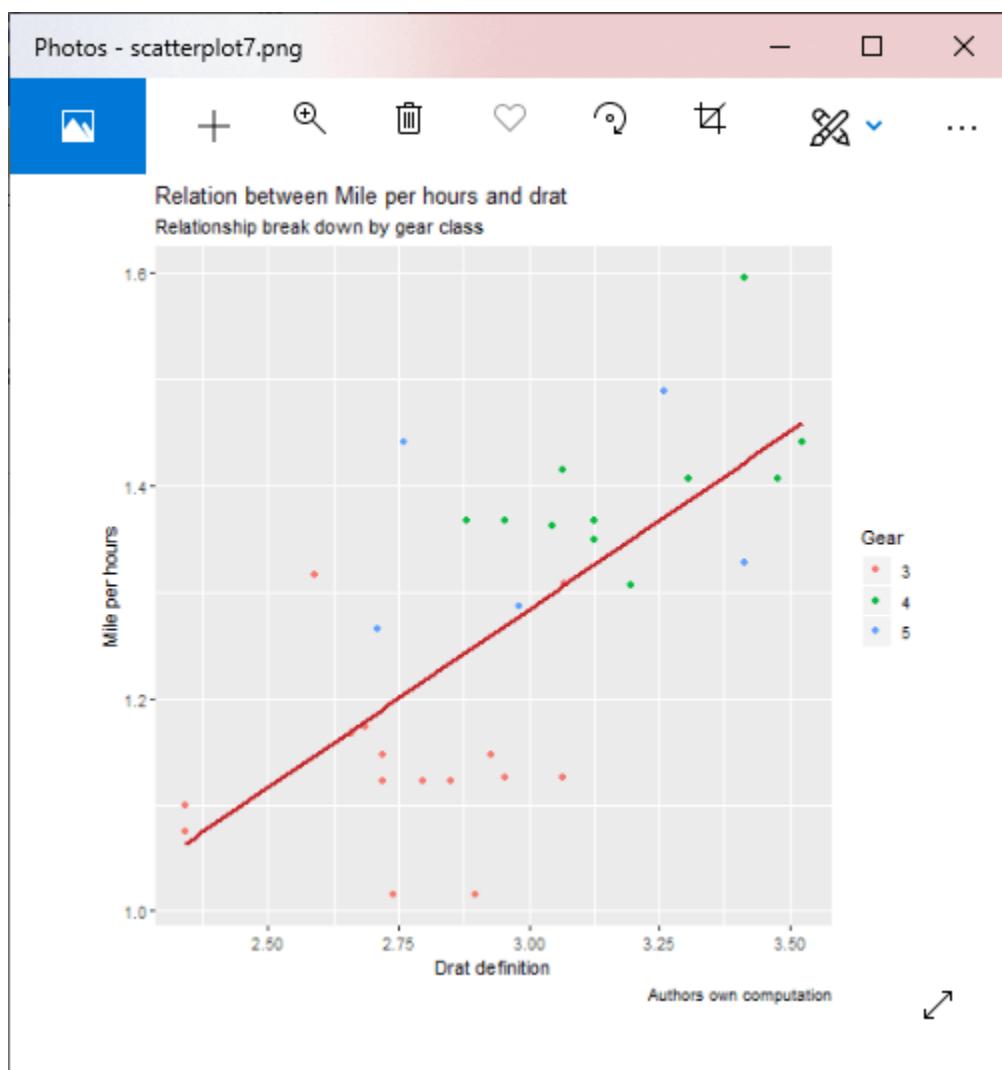
```
ggplot(mtcars, aes(x = log(mpg), y = log(drat))) +geom_point(aes(color = factor(gear))) +
```
8. stat_smooth(method = "lm", col = "#C42126", se = FALSE, size = 1)
9. #in above example lm is used for linear regression and se stands for standard error.
10. #Adding title with dynamic name
11. new_graph + labs(

```

12.   x = "Drat definition",
13.   y = "Mile per hours",
14.   color = "Gear",
15.   title = "Relation between Mile per hours and drat",
16.   subtitle = "Relationship break down by gear class",
17.   caption = "Authors own computation"
18.)
19.# Saving the file.
20.dev.off()

```

Output:

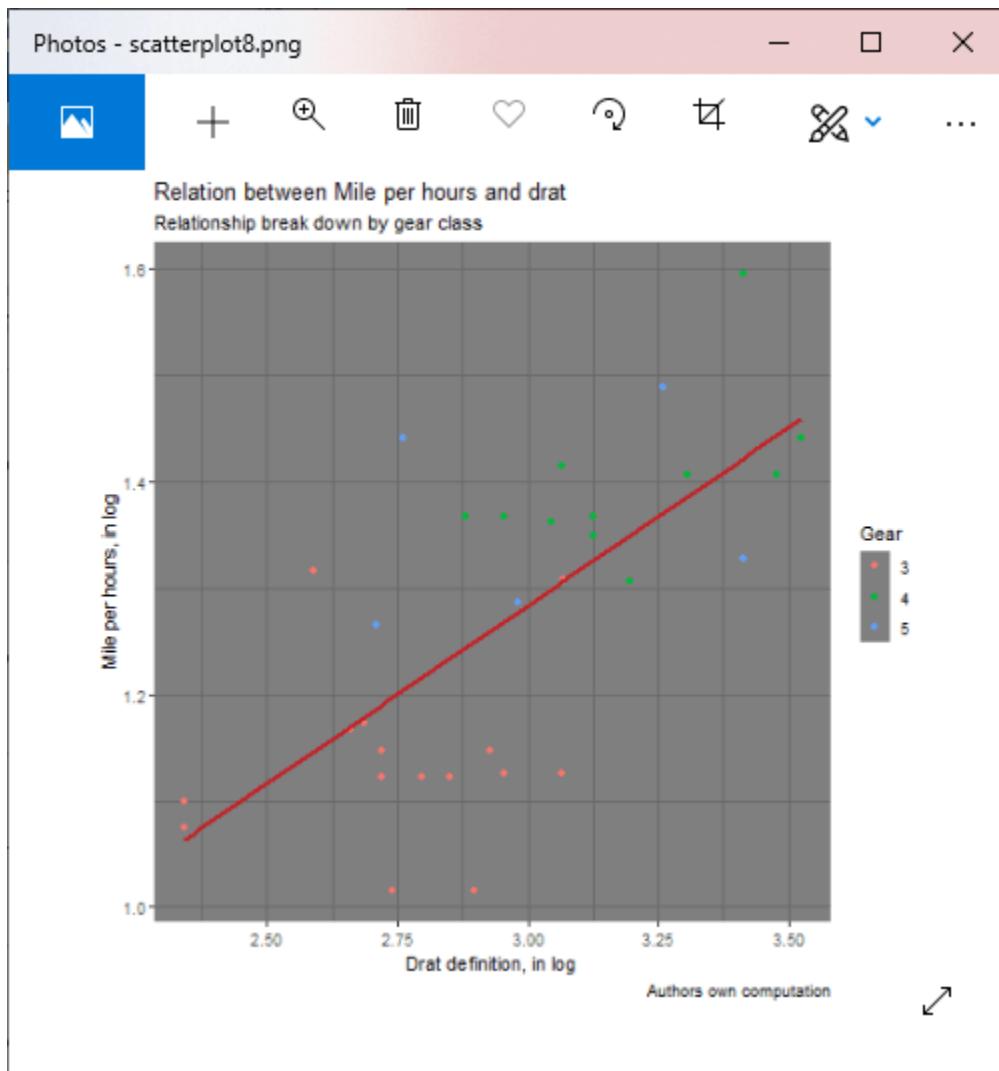


Example 8: Adding theme

1. #Loading ggplot2 package
2. library(ggplot2)

```
3. # Giving a name to the chart file.  
4. png(file = "scatterplot8.png")  
5. #Creating scatterplot with fitted values.  
6. # An additional function stst_smooth is used for linear regression.  
7. new_graph<-  
  ggplot(mtcars, aes(x = log(mpg), y = log(drat))) +geom_point(aes(color = fact  
or(gear))) +  
8. stat_smooth(method = "lm",col = "#C42126",se = FALSE,size = 1)  
9. #in above example lm is used for linear regression and se stands for standard  
error.  
10. #Adding title with dynamic name  
11. new_graph+  
12. theme_dark() +  
13.       labs(  
14.           x = "Drat definition, in log",  
15.           y = "Mile per hours, in log",  
16.           color = "Gear",  
17.           title = "Relation between Mile per hours and drat",  
18.           subtitle = "Relationship break down by gear class",  
19.           caption = "Authors own computation"  
20.       )  
21. # Saving the file.  
22. dev.off()
```

Output:



R Statistics

R - Mean, Median and Mode

Statistical analysis in R is performed by using many in-built functions. Most of these functions are part of the R base package. These functions take R vector as an input along with the arguments and give the result.

The functions we are discussing in this chapter are mean, median and mode.

Mean

It is calculated by taking the sum of the values and dividing with the number of values in a data series.

The function **mean()** is used to calculate this in R.

Syntax

The basic syntax for calculating mean in R is –

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

Following is the description of the parameters used –

- **x** is the input vector.
- **trim** is used to drop some observations from both end of the sorted vector.
- **na.rm** is used to remove the missing values from the input vector.

Example

[Live Demo](#)

```
# Create a vector.  
x <- c(12,7,3,4.2,18,2,54,-21,8,-5)  
  
# Find Mean.  
result<- mean(x)  
print(result)
```

When we execute the above code, it produces the following result –

```
[1] 8.22
```

Applying Trim Option

When trim parameter is supplied, the values in the vector get sorted and then the required numbers of observations are dropped from calculating the mean.

When **trim = 0.3**, 3 values from each end will be dropped from the calculations to find mean.

In this case the sorted vector is (-21, -5, 2, 3, 4.2, 7, 8, 12, 18, 54) and the values removed from the vector for calculating mean are (-21,-5,2) from left and (12,18,54) from right.

[Live Demo](#)

```
# Create a vector.  
x <- c(12,7,3,4.2,18,2,54,-21,8,-5)  
  
# Find Mean.  
result<- mean(x,trim = 0.3)  
print(result)
```

When we execute the above code, it produces the following result –

```
[1] 5.55
```

Applying NA Option

If there are missing values, then the mean function returns NA.

To drop the missing values from the calculation use na.rm = TRUE. which means remove the NA values.

[Live Demo](#)

```
# Create a vector.  
x <- c(12,7,3,4.2,18,2,54,-21,8,-5,NA)  
  
# Find mean.  
result <- mean(x)  
print(result.mean)  
  
# Find mean dropping NA values.  
result<- mean(x,na.rm = TRUE)  
print(result)
```

When we execute the above code, it produces the following result –

```
[1] NA  
[1] 8.22
```

Median

The middle most value in a data series is called the median. The **median()** function is used in R to calculate this value.

Syntax

The basic syntax for calculating median in R is –

```
median(x, na.rm = FALSE)
```

Following is the description of the parameters used –

- **x** is the input vector.
- **na.rm** is used to remove the missing values from the input vector.

Example

[Live Demo](#)

```
#-----median with odd values
```

#The middle most value in a data series is called the median. The median() function is used in R to calculate this value.

```
v <- c(10,15,5,7,20)

#first sort the vector into ascending way

# Calculate the mode using the user function.

result <- median(v)

print(result)
```

#-----median with Even values

```
v <- c(10,20,30,40,50,60)

#first sort the vector into ascending way

# Calculate the mode using the user function.

result <- median(v)

print(result)
```

```
# Create the vector.
x <- c(12,7,3,4,2,18,2,54,-21,8,-5)

# Find the median.
result <- median(x)
print(result)
```

When we execute the above code, it produces the following result –

```
[1] 5.6
```

Mode

The mode is the value that has highest number of occurrences in a set of data. Unlike mean and median, mode can have both numeric and character data.

R does not have a standard in-built function to calculate mode. So we create a user function to calculate mode of a data set in R. This function takes the vector as input and gives the mode value as output.

Example

[Live Demo](#)

```
library(modeest)
v <- c(2,1,2,3,1,2,3,4,1,5,5,3,2,3)
# Calculate the mode using the user function.
result <- mfv(v)
print(result)
```

```
# Create the function.
getmode <- function(v) {
  uniqv <- unique(v)
  uniqv[which.max(tabulate(match(v, uniqv)))]
}

# Create the vector with numbers.
v <- c(2,1,2,3,1,2,3,4,1,5,5,3,2,3)

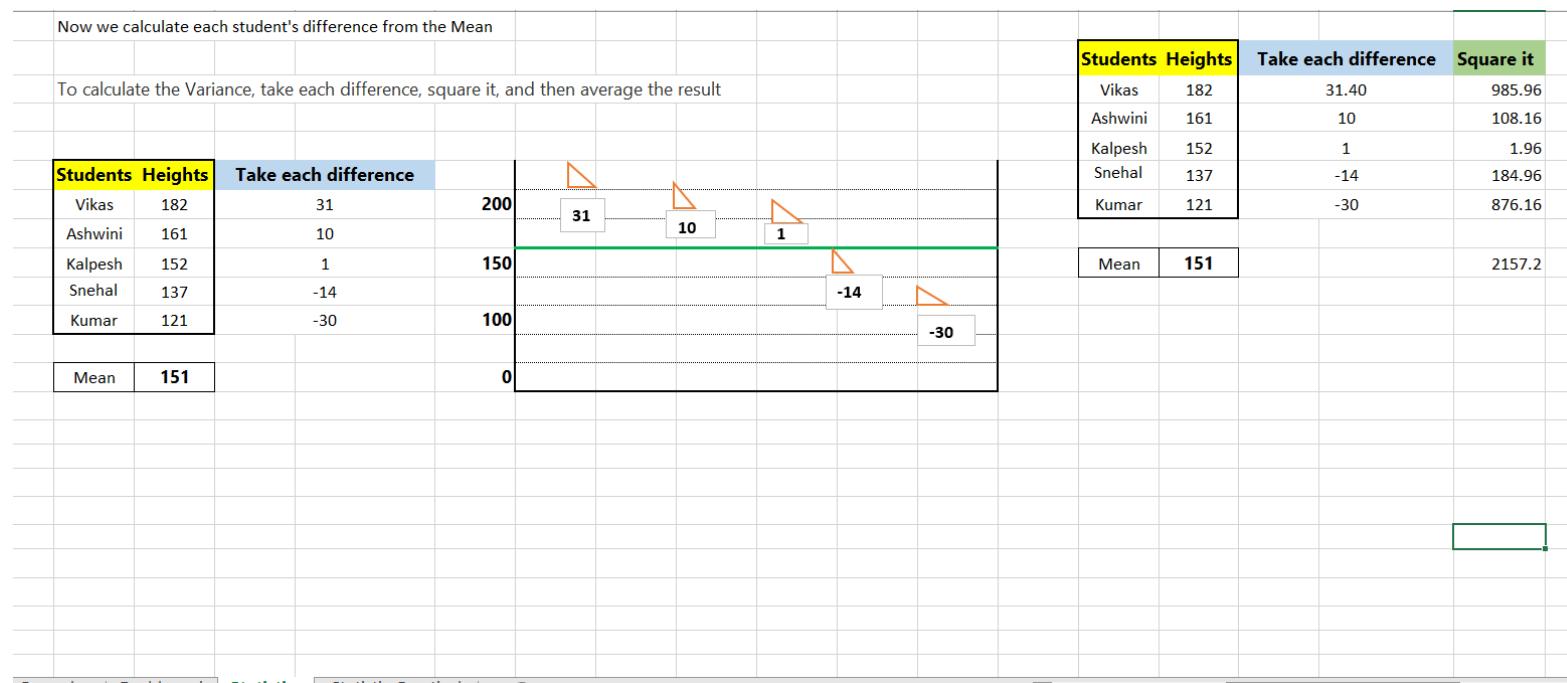
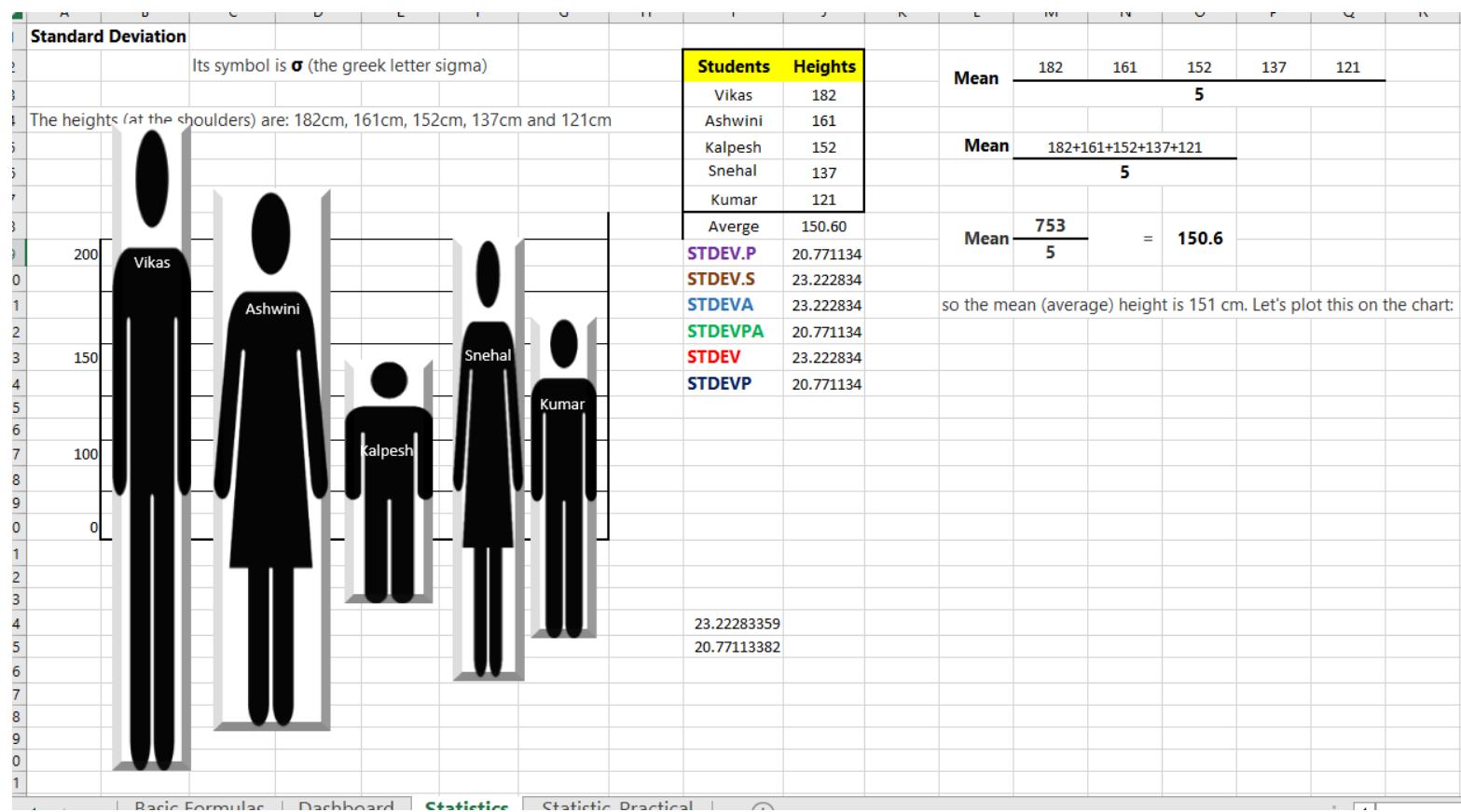
# Calculate the mode using the user function.
result <- getmode(v)
print(result)

# Create the vector with characters.
charv <- c("o","it","the","it","it")

# Calculate the mode using the user function.
result <- getmode(charv)
print(result)
```

When we execute the above code, it produces the following result –

```
[1] 2
[1] "it"
```



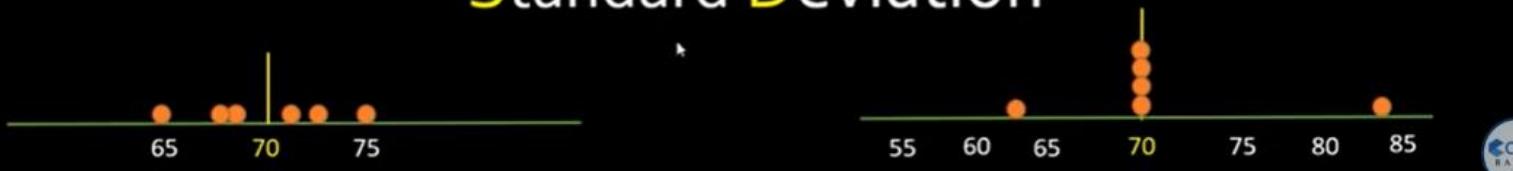
Average = 70

Average = 70

Name	Score	Abs (Score – Avg)	Abs (Score – Avg) ²
Mohan	75	5	25
Andrea	72	2	4
Sofia	68	2	4
Joe	65	5	25
Virat	67	3	9
Abdul	73	3	9
Avg		12.66	
\sqrt{Avg}		3.55	

Name	Score	Abs (Score – Avg)	Abs (Score – Avg) ²
Mohan	83	13	169
Andrea	70	0	0
Sofia	70	0	0
Joe	63	7	49
Virat	70	0	0
Abdul	70	0	0
Avg		36.33	
\sqrt{Avg}		6.02	

Standard Deviation



$$\sigma = \sqrt{\frac{\sum(x_i - \mu)^2}{N}}$$

σ = population standard deviation

N = the size of the population

x_i = each value from the population

μ = the population mean

From the web

To find the **standard deviation**, we take the square root of the variance. From learning that **SD = 13.31**, we can say that each score deviates from the mean by 13.31 points on average. Sep 17, 2020

```
Student<-c('Vikas','Ashwani','Kalpesh','Snehal','Kumar')

Height<-c(182,161,152,137,121)

mn<-mean(Height)

paste("Mean is ",mn)

std<-sd(Height)

paste("Standard Deviation ",std)
```

Linear Regression

linear regress only has one independent variable impacting the slope of the relationship, multiple regression incorporates multiple independent variables.

Linear regression is used to predict the value of an outcome variable y on the basis of one or more input predictor variables x . In other words, linear regression is used to establish a linear relationship between the predictor and response variables.

In linear regression, predictor and response variables are related through an equation in which the exponent of both these variables is 1. Mathematically, a linear relationship denotes a straight line, when plotted as a graph.

There is the following general mathematical equation for linear regression:

$$1. \quad y = ax + b$$

Here,

- y is a response variable.
- x is a predictor variable.
- a and b are constants that are called the coefficients.

Steps for establishing the Regression

The prediction of the weight of a person when his height is known, is a simple example of regression. To predict the weight, we need to have a relationship between the height and weight of a person.

There are the following steps to create the relationship:

1. In the first step, we carry out the experiment of gathering a sample of observed values of height and weight.
2. After that, we create a relationship model using the lm() function of R.
3. Next, we will find the coefficient with the help of the model and create the mathematical equation using this coefficient.
4. We will get the summary of the relationship model to understand the average error in prediction, known as residuals.
5. At last, we use the predict() function to predict the weight of the new person.

There is the following syntax of lm() function:

1. lm(formula,data)

Here,

S.No	Parameters	Description
1.	Formula	It is a symbol that presents the relationship between x and y.
2.	Data	It is a vector on which we will apply the formula.

Creating Relationship Model and Getting the Coefficients

Let's start performing the second and third steps, i.e., creating a relationship model and getting the coefficients. We will use the lm() function and pass the x and y input vectors and store the result in a variable named **relationship_model**.

Example

```

1. #Creating input vector for lm() function
x <- c(141, 134, 178, 156, 108, 116, 119, 143, 162, 130)
y <- c(62, 85, 56, 21, 47, 17, 76, 92, 62, 58)
# Applying the lm() function.
relationship_model<- lm(y~x)
#Printing the coefficient
print(relationship_model)

```

Output:

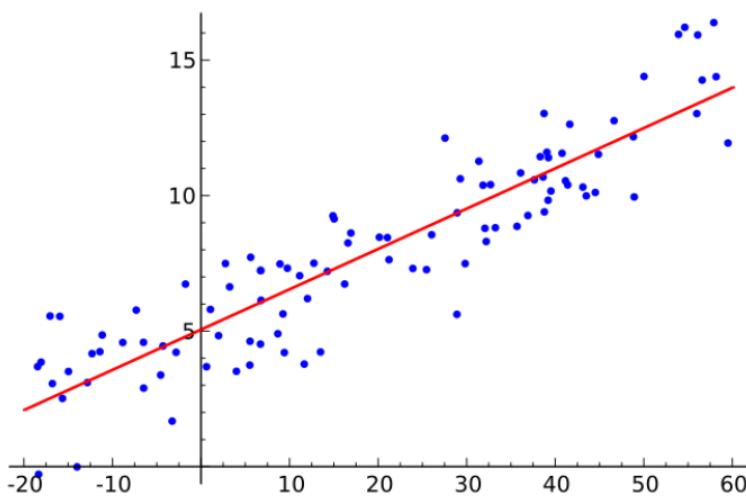
```
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)          x
47.50833      0.07276
```

Getting Summary of Relationship Model

We will use the summary() function to get a summary of the relationship model. Let's see an example to understand the use of the summary() function.

<p>The table shows the relation between the variables x and y. Find the equation of the regression line in the form $\hat{y} = a + bx$. Approximate a and b to 3 decimal places.</p>						
x	10	22	22	13	16	21
y	25	18	24	25	12	17
$\hat{y} = a + bx$ $s_{xx} = \sum x^2 - \frac{(\sum x)^2}{n}$						$a \approx 26.678, b \approx -0.376$
$b = \frac{s_{xy}}{s_{xx}}$ $s_{xy} = \sum xy - \frac{\sum x \sum y}{n}$						$\hat{y} = -0.376x + 26.678$
$a = \bar{y} - b\bar{x}$ $\bar{x} = \frac{\sum x}{n}$ $n=6$						$b = -\frac{74}{197} = -0.3756\dots$
$\bar{y} = \frac{\sum y}{n}$						$\bar{x} = \frac{104}{6} = \frac{52}{3}, \bar{y} = \frac{121}{6}$
$\sum x = 104$ $a = \frac{121}{6} + \left(\frac{74}{197}\right)\left(\frac{52}{3}\right) = \frac{121}{6} + \frac{3848}{591}$						$= 26.6776\dots$
$\sum y = 121$ $\sum x^2 = 1934$ $\sum xy = 2048$						



```
x<-c(10,22,22,13,16,21)
```

```
y<-c(25,18,24,25,12,17)
```

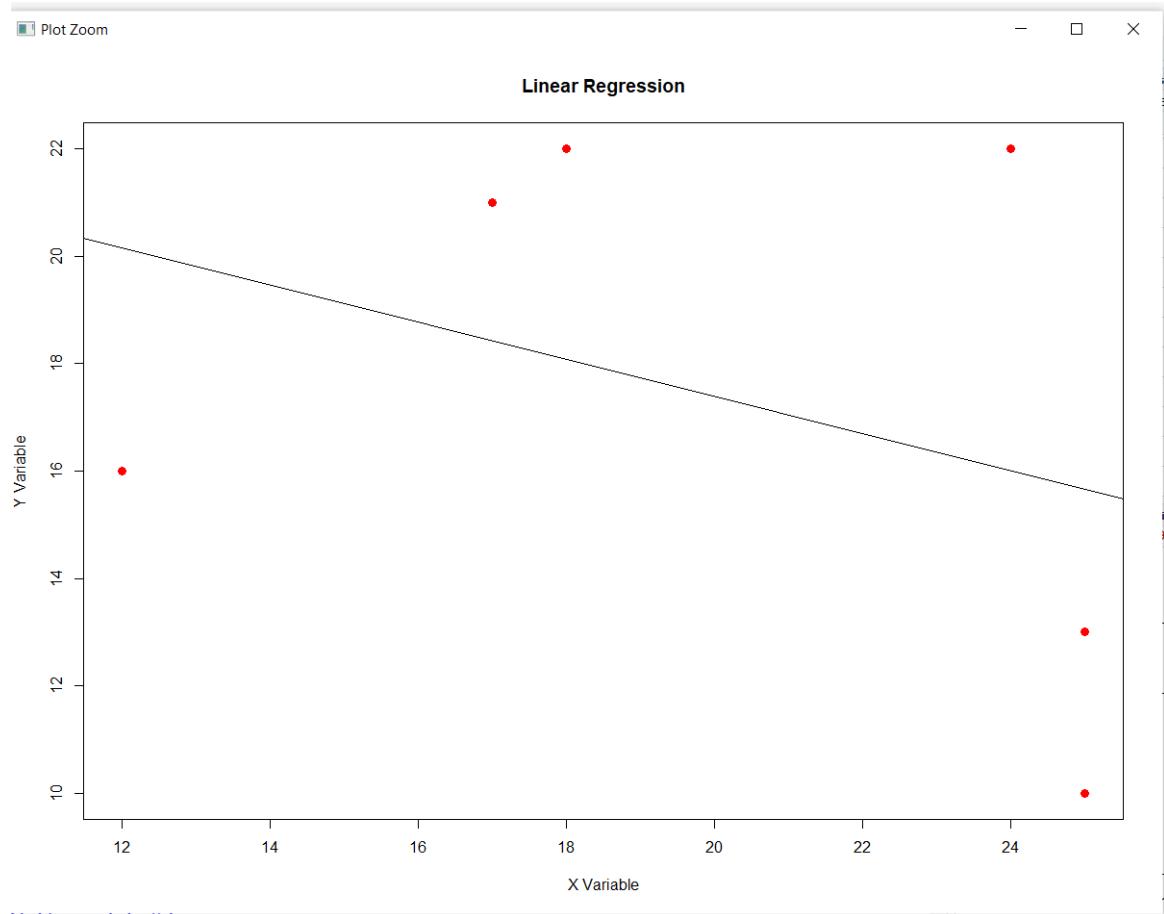
```

relationship_model<- lm(y~x)

print(relationship_model)

plot(y,x,col = "red",main = "Linear Regression",abline(lm(x~y)),cex = 1.3,pch = 16,xlab =
"X Variable",ylab = "Y Variable")

```



Example

1. #Creating input vector **for** lm() function
2. x <- c(141, 134, 178, 156, 108, 116, 119, 143, 162, 130)
3. y <- c(62, 85, 56, 21, 47, 17, 76, 92, 62, 58)
- 4.
5. # Applying the lm() function.
6. relationship_model<- lm(y~x)
- 7.
8. #Printing the coefficient
9. print(summary(relationship_model))

Output:

```
Call:
lm(formula = y ~ x)

Residuals:
    Min      1Q  Median      3Q     Max 
-38.948 -7.390  1.869  15.933  34.087 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept) 47.50833   55.18118   0.861   0.414    
x             0.07276    0.39342   0.185   0.858    
                                                        
Residual standard error: 25.96 on 8 degrees of freedom
Multiple R-squared:  0.004257, Adjusted R-squared:  -0.1202 
F-statistic: 0.0342 on 1 and 8 DF,  p-value: 0.8579
```

The predict() Function

Now, we will predict the weight of new persons with the help of the predict() function. There is the following syntax of predict function:

1. `predict(object, newdata)`

Here,

S.No	Parameter	Description
1.	object	It is the formula that we have already created using the lm() function.
2.	Newdata	It is the vector that contains the new value for the predictor variable.

Example

1. #Creating input vector **for** lm() function
2. `x <- c(141, 134, 178, 156, 108, 116, 119, 143, 162, 130)`
3. `y <- c(62, 85, 56, 21, 47, 17, 76, 92, 62, 58)`
- 4.
5. # Applying the lm() function.
6. `relationship_model<- lm(y~x)`
- 7.
8. # Finding the weight of a person with height 170.
9. `z <- data.frame(x = 160)`
10. `predict_result<- predict(relationship_model,z)`

```
11. print(predict_result)
```

Output:

```
1  
59.14977
```

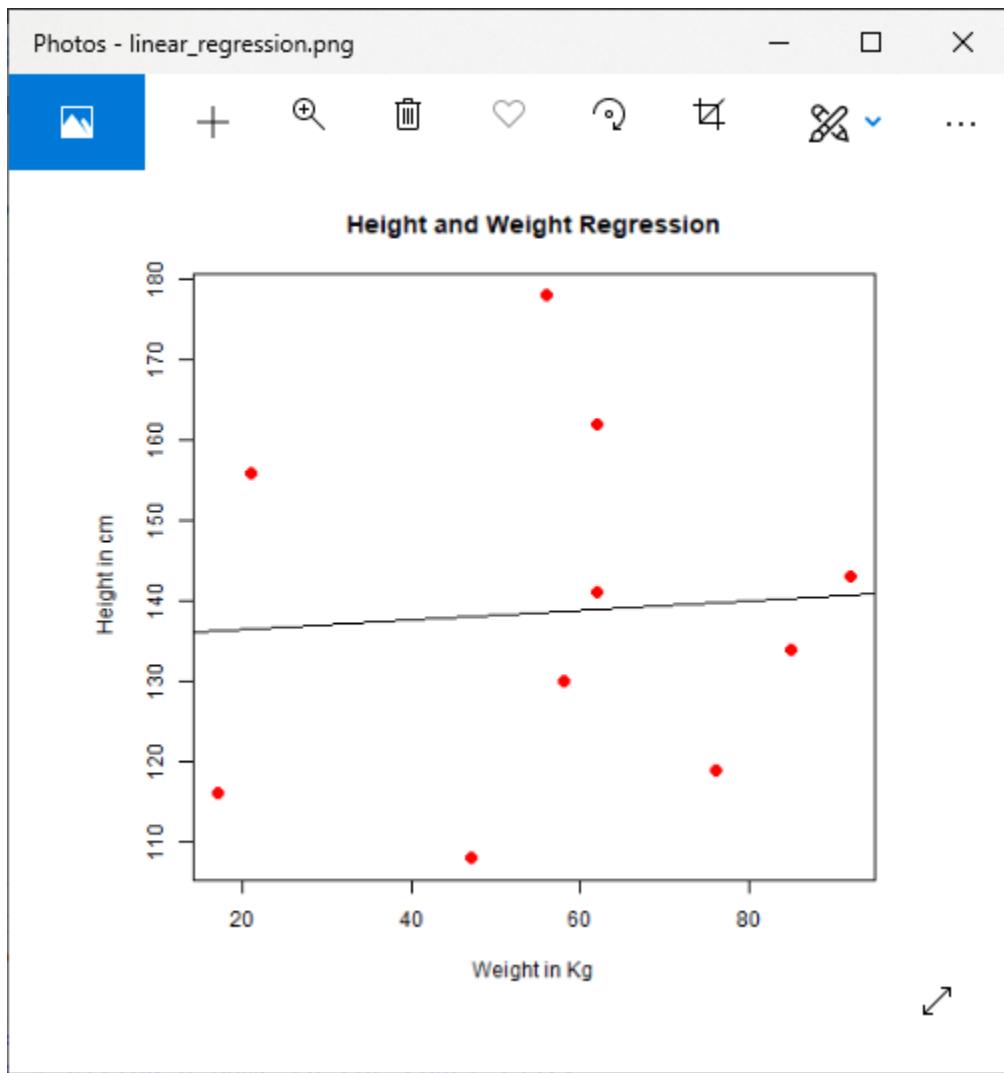
Plotting Regression

Now, we plot out prediction results with the help of the `plot()` function. This function takes parameter `x` and `y` as an input vector and many more arguments.

Example

1. #Creating input vector **for lm()** function
2. `x <- c(141, 134, 178, 156, 108, 116, 119, 143, 162, 130)`
3. `y <- c(62, 85, 56, 21, 47, 17, 76, 92, 62, 58)`
4. `relationship_model<- lm(y~x)`
5. # Giving a name to the chart file.
6. `png(file = "linear_regression.png")`
7. # Plotting the chart.
8. `plot(y,x,col = "red",main = "Height and Weight Regression",abline(lm(x~y)),cex = 1.3,pch = 16,xlab = "Weight in Kg",ylab = "Height in cm")`
9. # Saving the file.
10. `dev.off()`

Output:



R-Multiple Linear Regression

Multiple linear regression is the extension of the simple linear regression, which is used to predict the outcome variable (y) based on multiple distinct predictor variables (x). With the help of three predictor variables (x_1, x_2, x_3), the prediction of y is expressed using the following equation:

$$y = b_0 + b_1 * x_1 + b_2 * x_2 + b_3 * x_3$$

The "b" values represent the regression weights. They measure the association between the outcome and the predictor variables."

Or

Multiple linear regression is the extension of linear regression in the relationship between more than two variables. In simple linear regression, we have one predictor

and one response variable. But in multiple regressions, we have more than one predictor variable and one response variable.

There is the following general mathematical equation for multiple regression -

$$y = b_0 + b_1 * x_1 + b_2 * x_2 + b_3 * x_3 + \dots + b_n * x_n$$

Here,

- **y** is a response variable.
- **b0, b1, b2...bn** are the coefficients.
- **x1, x2, ...xn** are the predictor variables.

In R, we create the regression model with the help of the **lm()** function. The model will determine the value of the coefficients with the help of the input data. We can predict the value of the response variable for the set of predictor variables using these coefficients.

There is the following syntax of lm() function in multiple regression

1. `lm(y ~ x1+x2+x3..., data)`

Before proceeding further, we first create our data for multiple regression. We will use the "mtcars" dataset present in the R environment. The main task of the model is to create the relationship between the "mpg" as a response variable with "wt", "disp" and "hp" as predictor variables.

For this purpose, we will create a subset of these variables from the "mtcars" dataset.

```
data<-mtcars[,c("mpg","wt","disp","hp")]
print(head(input))
```

Output:

```
Rterm (64-bit)
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

> data<-mtcars[,c("mpg","wt","disp","hp")]
> print(head(data))
      mpg   wt disp hp
Mazda RX4    21.0 2.620 160 110
Mazda RX4 Wag 21.0 2.875 160 110
Datsun 710   22.8 2.320 108  93
Hornet 4 Drive 21.4 3.215 258 110
Hornet Sportabout 18.7 3.440 360 175
Valiant     18.1 3.460 225 105
>
```

Creating Relationship Model and finding Coefficient

Now, we will use the data which we have created before to create the Relationship Model. We will use the `lm()` function, which takes two parameters i.e., formula and data. Let's start understanding how the `lm()` function is used to create the Relationship Model.

Example

1. #Creating input data.
2. input <- mtcars[,c("mpg","wt","disp","hp")]
3. # Creating the relationship model.
4. Model <- lm(mpg~wt+disp+hp, data = input)
5. # Showing the Model.
6. print(Model)

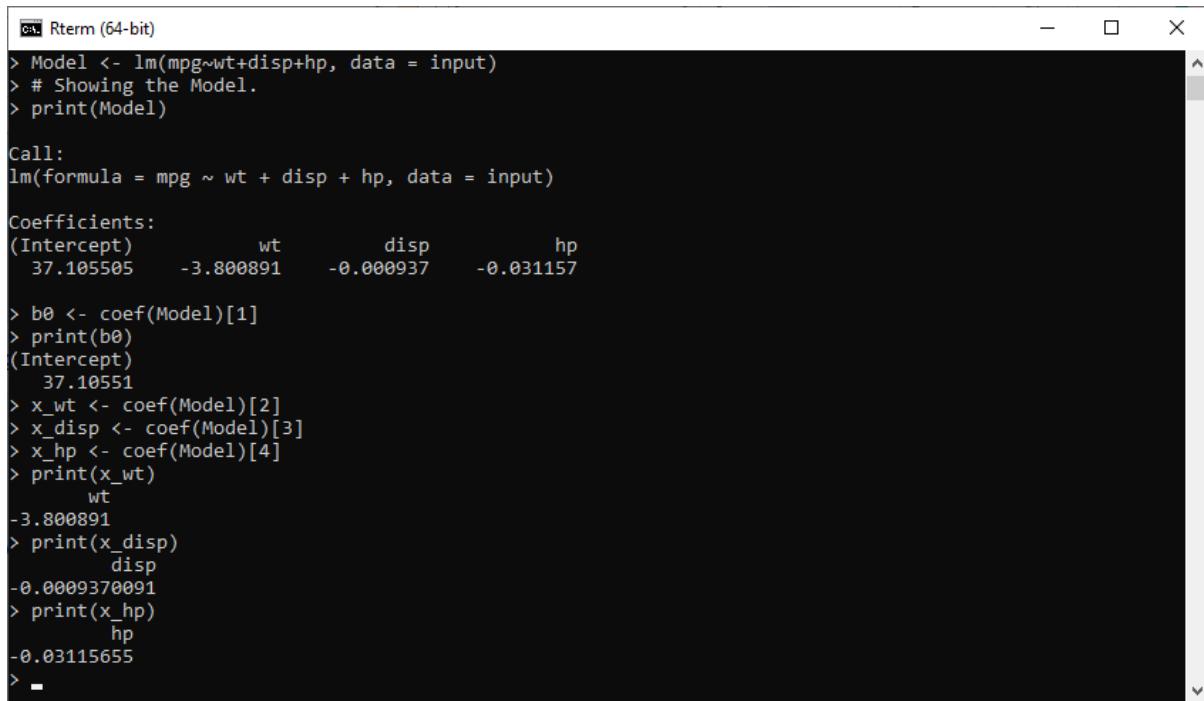
Output:

From the above output it is clear that our model is successfully setup. Now, our next step is to find the coefficient with the help of the model.

```
b0<- coef(Model) [1]
print(b0)
x_wt<- coef(Model) [2]
x_disp<- coef(Model) [3]
x_hp<- coef(Model) [4]
print(x_wt)
```

```
print(x_disp)
print(x_hp)
```

Output:



```
Rterm (64-bit)
> Model <- lm(mpg~wt+disp+hp, data = input)
> # Showing the Model.
> print(Model)

Call:
lm(formula = mpg ~ wt + disp + hp, data = input)

Coefficients:
(Intercept)          wt          disp          hp
 37.105505     -3.800891    -0.000937    -0.031157

> b0 <- coef(Model)[1]
> print(b0)
(Intercept)
37.10551
> x_wt <- coef(Model)[2]
> x_disp <- coef(Model)[3]
> x_hp <- coef(Model)[4]
> print(x_wt)
      wt
-3.800891
> print(x_disp)
      disp
-0.0009370091
> print(x_hp)
      hp
-0.03115655
> -
```

The equation for the Regression Model

Now, we have coefficient values and intercept. Let's start creating a mathematical equation that we will apply for predicting new values. First, we will create an equation, and then we use the equation to predict the mileage when a new set of values for weight, displacement, and horsepower is provided.

Let's see an example in which we predict the mileage for a car with weight=2.51, disp=211 and hp=82.

Example

1. #Creating equation for predicting new values.
2. $y=b0+x_wt*x1+x_disp*x2+x_hp*x3\$
3. #Applying equation for prediction new values
4. $y=b0+x_wt*2.51+x_disp*211+x_hp*82$

Output:

```
Command Prompt
C:\Users\ajeet\R>Rscript multi_reg.R

Call:
lm(formula = mpg ~ wt + disp + hp, data = input)

Coefficients:
(Intercept)          wt          disp          hp      
 37.105505     -3.800891    -0.000937    -0.031157

C:\Users\ajeet\R>
```

```
x<-c(10,22,22,13,16,21)
y<-c(25,18,24,25,12,17)
relationship_model<- lm(y~x)
print(relationship_model)
print(summary(relationship_model))

plot(y,x,col = "red",main = "Linear Regression",abline(lm(x~y)),cex = 1.3,pch = 16,xlab = "X
Variable",ylab = "Y Variable")

#-----Linear regression with Multiple X variable or predictor-----
x1<-c(10,22,22,13,16,21)
x2<-c(5,10,15,20,25,30)
y<-c(25,18,24,25,12,17)
relationship_model<- lm(y~x1+x2)
print(relationship_model)

plot(y,x1+x2,col = "red",main = "Linear Regression",abline(lm(x1+x2~y)),cex = 1.3,pch = 16,xlab = "X
Variable",ylab = "Y Variable")
```

R-Logistic Regression

The Logistic Regression is a regression model in which the response variable (dependent variable) has categorical values such as True/False or 0/1. It actually measures the probability of a binary response as the value of response variable based on the mathematical equation relating it with the predictor variables.

The general mathematical equation for logistic regression is –

$$P = \frac{1}{1 + e^{-(.5596 + 1.2528X)}}$$

$$y = 1/(1+e^{-(a+b_1x_1+b_2x_2+b_3x_3+\dots)})$$

Following is the description of the parameters used –

- **y** is the response variable.
- **x** is the predictor variable.
- **a** and **b** are the coefficients which are numeric constants.

The function used to create the regression model is the **glm()** function.

Syntax

The basic syntax for **glm()** function in logistic regression is –

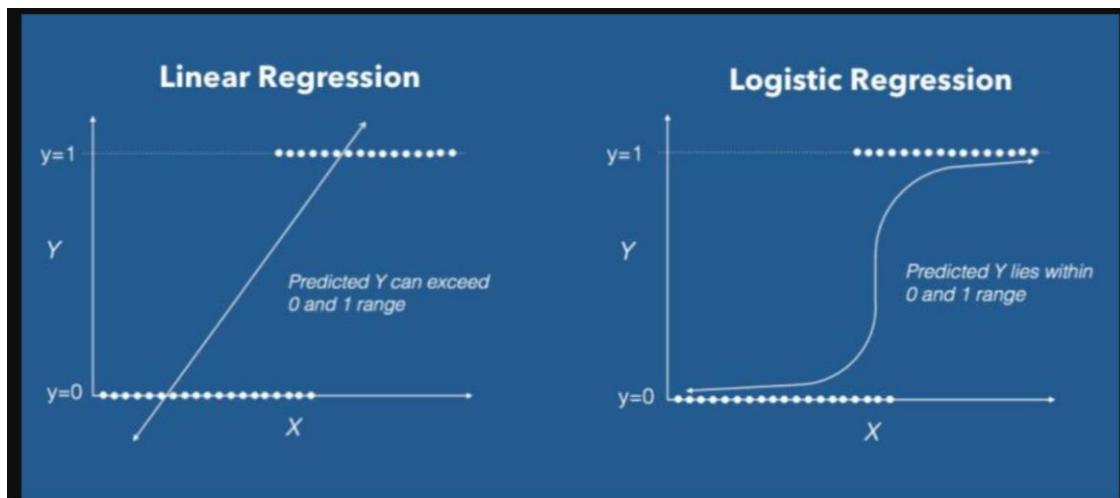
$$\text{glm}(\text{formula}, \text{data}, \text{family})$$

Following is the description of the parameters used –

- **formula** is the symbol presenting the relationship between the variables.
- **data** is the data set giving the values of these variables.
- **family** is R object to specify the details of the model. Its value is binomial for logistic regression.

Example

The in-built data set "mtcars" describes different models of a car with their various engine specifications. In "mtcars" data set, the transmission mode (automatic or manual) is described by the column am which is a binary value (0 or 1). We can create a logistic regression model between the columns "am" and 3 other columns - hp, wt and cyl.



1. `glm(formula, data, family)`

Here,

S.No	Parameter	Description
1.	<code>formula</code>	It is a symbol which represents the relationship b/w the variables.
2.	<code>data</code>	It is the dataset giving the values of the variables.
3.	<code>family</code>	An R object which specifies the details of the model, and its value is binomial.

Building Logistic Regression

The in-built data set "mtcars" describes various models of the car with their different engine specifications. In the "mtcars" data set, the transmission mode is described by the column "am", which is a binary value (0 or 1). We can construct a logistic regression model between column "am" and three other columns - hp, wt, and cyl.

Let's see an example to understand how the `glm` function is used to create logistic regression and how we can use the `summary` function to find a summary for the analysis.

In our example, we will use the dataset "BreastCancer" available in the R environment. To use it, we first need to install "mlbench" and "caret" packages.

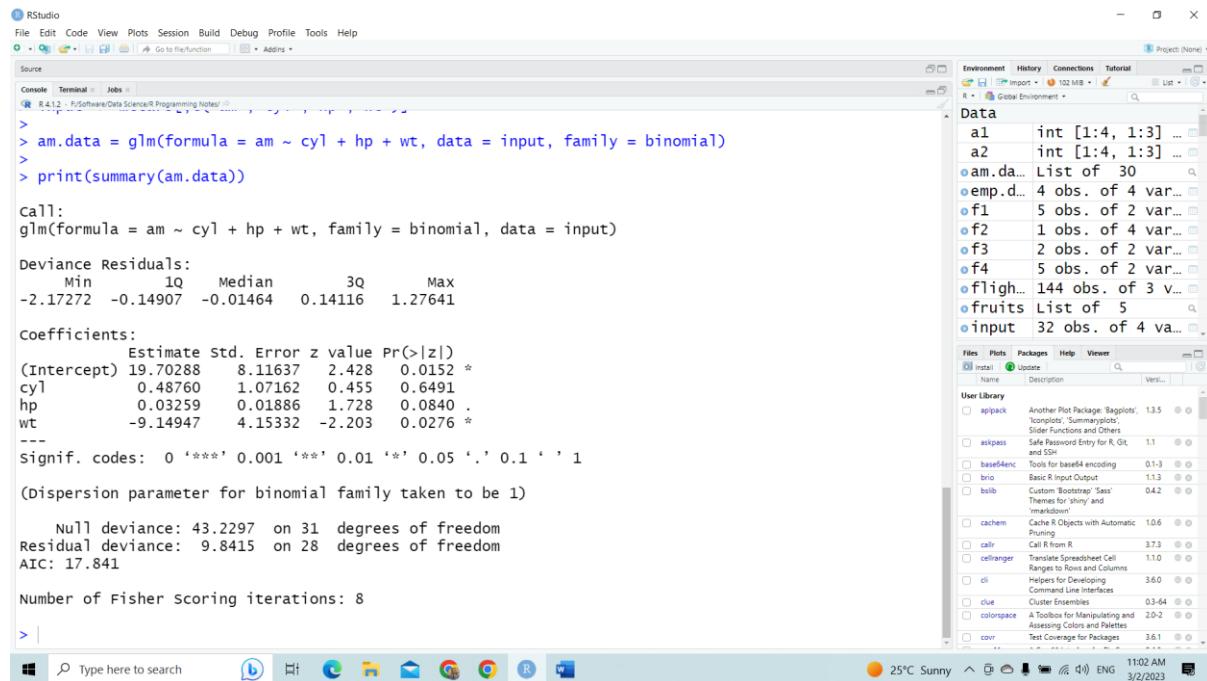
Example

```
input <- mtcars[,c("am","cyl","hp","wt")]
```

```
am.data = glm(formula = am ~ cyl + hp + wt, data = input, family = binomial)
```

```
print(summary(am.data))
```

Output:



The screenshot shows the RStudio interface with the following details:

- Console Tab:** Displays the R code and its output:

```
> am.data = glm(formula = am ~ cyl + hp + wt, data = input, family = binomial)
> print(summary(am.data))

Call:
glm(formula = am ~ cyl + hp + wt, family = binomial, data = input)

Deviance Residuals:
    Min      1Q  Median      3Q     Max 
-2.17272 -0.14907 -0.01464  0.14116  1.27641 

Coefficients:
            Estimate Std. Error z value Pr(>|z|)    
(Intercept) 19.70288   8.11637  2.428   0.0152 *  
cyl          0.48760   1.07162  0.455   0.6491    
hp           0.03259   0.01886  1.728   0.0840 .  
wt          -9.14947   4.15332 -2.203   0.0276 *  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 43.2297 on 31 degrees of freedom
Residual deviance: 9.8415 on 28 degrees of freedom
AIC: 17.841

Number of Fisher Scoring iterations: 8
>
```
- Data View:** Shows the structure of the data objects:

Object	Type	Dimensions
a1	int	[1:4, 1:3]
a2	int	[1:4, 1:3]
am.da...	List	of 30
emp.d...	4 obs.	of 4 var...
f1	5 obs.	of 2 var...
f2	1 obs.	of 4 var...
f3	2 obs.	of 2 var...
f4	5 obs.	of 2 var...
flight...	144 obs.	of 3 var...
fruits	List	of 5
input	32 obs.	of 4 var...
- Environment View:** Shows the global environment with various packages listed.

We now divide our data into training and test sets with training sets containing 70% data and test sets including the remaining percentages.

1. #Dividing dataset into training and test dataset.
2. set.seed(100)
3. #Creating partitioning.
4. Training_Ratio <- createDataPartition(b_canc\$Class, p=0.7, list = F)
5. #Creating training data.
6. Training_Data <- b_canc[Training_Ratio,]
7. str(Training_Data)
8. #Creating test data.
9. Test_Data <- b_canc[-Training_Ratio,]
10. str(Test_Data)

Output:

```
C:\Users\ajeet\R>Rscript logistic_regression.R
Loading required package: lattice
Loading required package: ggplot2
'data.frame': 479 obs. of 11 variables:
$ Id           : chr "1000025" "1002945" "1015425" "1016277" ...
$ Cl.thickness : Ord.factor w/ 10 levels "1"  
"2"  
"3"  
"4"  
... : 5 5 3 6 8 2 2 1 7 4 ...
$ Cell.size    : Ord.factor w/ 10 levels "1"  
"2"  
"3"  
"4"  
... : 1 4 1 8 10 1 1 1 4 1 ...
$ Cell.shape   : Ord.factor w/ 10 levels "1"  
"2"  
"3"  
"4"  
... : 1 4 1 8 10 2 1 1 6 1 ...
$ Marg.adhesion: Ord.factor w/ 10 levels "1"  
"2"  
"3"  
"4"  
... : 1 5 1 1 8 1 1 1 4 1 ...
$ Epith.c.size : Ord.factor w/ 10 levels "1"  
"2"  
"3"  
"4"  
... : 2 7 2 3 7 2 2 2 6 2 ...
$ Bare.nuclei  : Factor w/ 10 levels "1", "2", "3", "4", ... : 1 10 2 4 10 1 1 3 1 1 ...
$ Bl.cromatin  : Factor w/ 10 levels "1", "2", "3", "4", ... : 3 3 3 3 9 3 2 3 4 3 ...
$ Normal.nucleoli: Factor w/ 10 levels "1", "2", "3", "4", ... : 1 2 1 7 7 1 1 1 3 1 ...
$ Mitoses      : Factor w/ 9 levels "1", "2", "3", "4", ... : 1 1 1 1 1 1 1 1 1 ...
$ Class         : Factor w/ 2 levels "benign", "malignant": 1 1 1 1 2 1 1 1 2 1 ...
'data.frame': 204 obs. of 11 variables:
$ Id           : chr "1017023" "1018099" "1033078" "1033078" ...
$ Cl.thickness : Ord.factor w/ 10 levels "1"  
"2"  
"3"  
"4"  
... : 4 1 2 4 1 5 8 4 1 3 ...
$ Cell.size    : Ord.factor w/ 10 levels "1"  
"2"  
"3"  
"4"  
... : 1 1 1 2 1 3 7 1 1 2 ...
$ Cell.shape   : Ord.factor w/ 10 levels "1"  
"2"  
"3"  
"4"  
... : 1 1 1 1 1 3 5 1 1 1 ...
$ Marg.adhesion: Ord.factor w/ 10 levels "1"  
"2"  
"3"  
"4"  
... : 3 1 1 1 1 3 10 1 1 1 ...
$ Epith.c.size : Ord.factor w/ 10 levels "1"  
"2"  
"3"  
"4"  
... : 2 2 2 2 1 2 7 2 2 1 ...
$ Bare.nuclei  : Factor w/ 10 levels "1", "2", "3", "4", ... : 1 10 1 1 1 3 9 1 1 1 ...
$ Bl.cromatin  : Factor w/ 10 levels "1", "2", "3", "4", ... : 3 3 1 2 3 4 5 2 3 2 ...
$ Normal.nucleoli: Factor w/ 10 levels "1", "2", "3", "4", ... : 1 1 1 1 1 4 5 1 1 1 ...
$ Mitoses      : Factor w/ 9 levels "1", "2", "3", "4", ... : 1 1 5 1 1 1 4 1 1 1 ...
$ Class         : Factor w/ 2 levels "benign", "malignant": 1 1 1 1 1 2 2 1 1 1 ...
C:\Users\ajeet\R>
```

Now, we construct the logistic regression function with the help of `glm()` function. We pass the formula **Class~Cell.shape** as the first parameter and specifying the attribute family as "**binomial**" and use Training_data as the third parameter.

Example

1. #Creating Regression Model
2. `glm(Class ~ Cell.shape, family = "binomial", data = Training_Data)`

Output:

```
C:\Users\ajeet\R>Select Command Prompt
C:\Users\ajeet\R>Rscript logistic_regression.R
Loading required package: lattice
Loading required package: ggplot2

Call: glm(formula = Class ~ Cell.shape, family = "binomial", data = Training_Data)

Coefficients:
(Intercept) Cell.shape.L Cell.shape.Q Cell.shape.C Cell.shape^4
        4.1470     20.7653      7.3156      5.4212     -1.3400
Cell.shape^5 Cell.shape^6 Cell.shape^7 Cell.shape^8 Cell.shape^9
       -4.2334     -4.9429     -3.2306     -1.8817     -0.9361

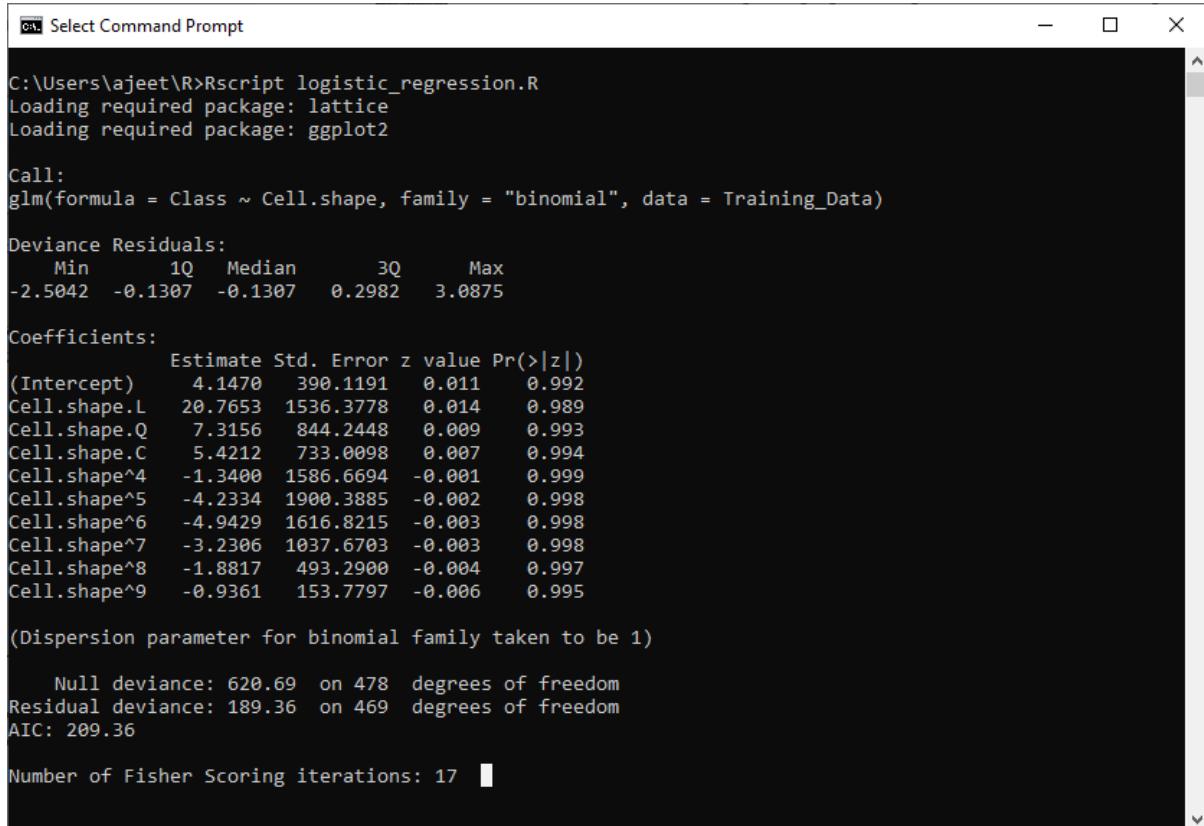
Degrees of Freedom: 478 Total (i.e. Null); 469 Residual
Null Deviance: 620.7
Residual Deviance: 189.4          AIC: 209.4

C:\Users\ajeet\R>
```

Now, use the summary function for analysis.

1. #Creating Regression Model
2. model<-**glm**(Class ~ Cell.shape, family="binomial", data = Training_Data)
3. #Using summary function
4. print(summary(model))

Output:



```
Windows PowerShell
Select Command Prompt
C:\Users\ajeet\R>Rscript logistic_regression.R
Loading required package: lattice
Loading required package: ggplot2

Call:
glm(formula = Class ~ Cell.shape, family = "binomial", data = Training_Data)

Deviance Residuals:
    Min      1Q  Median      3Q     Max 
-2.5042 -0.1307 -0.1307  0.2982  3.0875 

Coefficients:
            Estimate Std. Error z value Pr(>|z|)    
(Intercept)  4.1470   390.1191  0.011   0.992    
Cell.shape.L 20.7653  1536.3778  0.014   0.989    
Cell.shape.Q  7.3156   844.2448  0.009   0.993    
Cell.shape.C  5.4212   733.0098  0.007   0.994    
Cell.shape^4 -1.3400  1586.6694 -0.001   0.999    
Cell.shape^5 -4.2334  1900.3885 -0.002   0.998    
Cell.shape^6 -4.9429  1616.8215 -0.003   0.998    
Cell.shape^7 -3.2306  1037.6703 -0.003   0.998    
Cell.shape^8 -1.8817   493.2900 -0.004   0.997    
Cell.shape^9 -0.9361   153.7797 -0.006   0.995    

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 620.69 on 478 degrees of freedom
Residual deviance: 189.36 on 469 degrees of freedom
AIC: 209.36

Number of Fisher Scoring iterations: 17
```

R Poisson Regression

What is a Poisson distribution?

A Poisson distribution is defined as a discrete frequency distribution that gives the probability of the number of independent events that occur in the fixed time.

When do we use Poisson distribution?

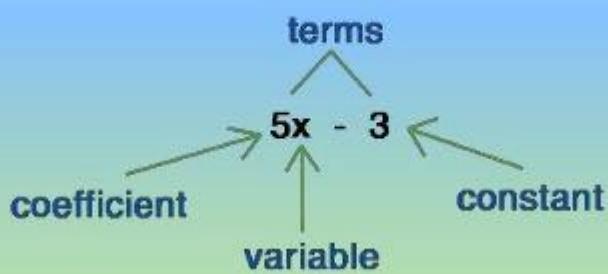
Poisson distribution is used when the independent events occurring at a constant rate within the given interval of time are provided.

What is the difference between the Poisson distribution and normal distribution?

Coefficients:

A number used to multiply a variable.

Example: $6z$ means 6 times z , and "z" is a variable, so 6 is a coefficient.



Coefficient



The diagram shows a polynomial expression $5x^2 + 8y + 2$ enclosed in a rounded orange box. Below the expression, there are two labels: "Coefficients" pointing to the terms $5x^2$ and $8y$, and "Constant" pointing to the term 2 .

The major difference between the Poisson distribution and the normal distribution is that the Poisson distribution is discrete whereas the normal distribution is continuous. If the mean of the Poisson distribution becomes larger, then the Poisson distribution is similar to the normal distribution.

The **Poisson Regression** model is used for modeling events where the outcomes are counts. Count data is a discrete data with non-negative integer values that count things, such as the number of people in line at the grocery store, or the number of times an event occurs during the given timeframe.

We can also define the **count data** as the rate data. So that it can express the number of times an event occurs within the timeframe as a raw count or as a rate. Poisson regression allows us to determine which explanatory variable (x values) influence a given response variable (y value, count, or a rate).

For example, poisson regression can be implemented by a grocery store to understand better, and predict the number of people in a row.

There is the following general mathematical equation for poisson regression:

Here,

S.No	Parameter	Description
1.	y	It is the response variable.
2.	a and b	These are the numeric coefficients.
3.	x	x is the predictor variable.

The poisson regression model is created with the help of the familiar function `glm()`.

Let's see an example in which we create the poisson regression model using `glm()` function. In this example, we have considered an in-built dataset "warpbreaks" that describe the tension(low, medium, or high), and the effect of wool type(A and B) on the number of wrap breaks per loom. We will consider wool "type" and "tension"as the predictor variables, and "breaks" is taken as the response variable.

Example

1. #Creating data for the poisson regression
2. `reg_data <- warpbreaks`
3. `print(head(reg_data))`

Output:

```
Microsoft Windows [Version 10.0.18362.239]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\ajeet>cd R

C:\Users\ajeet\R>Rscript poisson_regression.R
  breaks wool tension
1      26     A       L
2      30     A       L
3      54     A       L
4      25     A       L
5      70     A       L
6      52     A       L

C:\Users\ajeet\R>
```

Now, we will create the regression model with the help of the `glm()` function as:

1. #Creating Poisson Regression Model using `glm()` function

```
output_result <-  
  glm(formula = breaks ~ wool+tension, data = warpbreaks,family = poisson)  
output_result
```

Output:

```
C:\Users\ajeet\R>Rscript poisson_regression.R

Call:  glm(formula = breaks ~ wool + tension, family = poisson,  
          data = warpbreaks)

Coefficients:  
(Intercept)      woolB      tensionM      tensionH  
            3.6920     -0.2060      -0.3213      -0.5185

Degrees of Freedom: 53 Total (i.e. Null);  50 Residual  
Null Deviance: 297.4  
Residual Deviance: 210.4      AIC: 493.1

C:\Users\ajeet\R>
```

Now, let's use `summary()` function to find the summary of the model for data analysis.

1. #Using `summary` function
2. `print(summary(output_result))`

Output:

```
Command Prompt - □ X

C:\Users\ajeet\R>Rscript poisson_regression.R

Call:
glm(formula = breaks ~ wool + tension, family = poisson, data =
warpbreaks)

Deviance Residuals:
    Min      1Q  Median      3Q     Max 
-3.6871 -1.6503 -0.4269  1.1902  4.2616 

Coefficients:
            Estimate Std. Error z value Pr(>|z|)    
(Intercept) 3.69196   0.04541  81.302 < 2e-16 ***
woolB       -0.20599   0.05157  -3.994 6.49e-05 ***
tensionM    -0.32132   0.06027  -5.332 9.73e-08 ***
tensionH    -0.51849   0.06396  -8.107 5.21e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

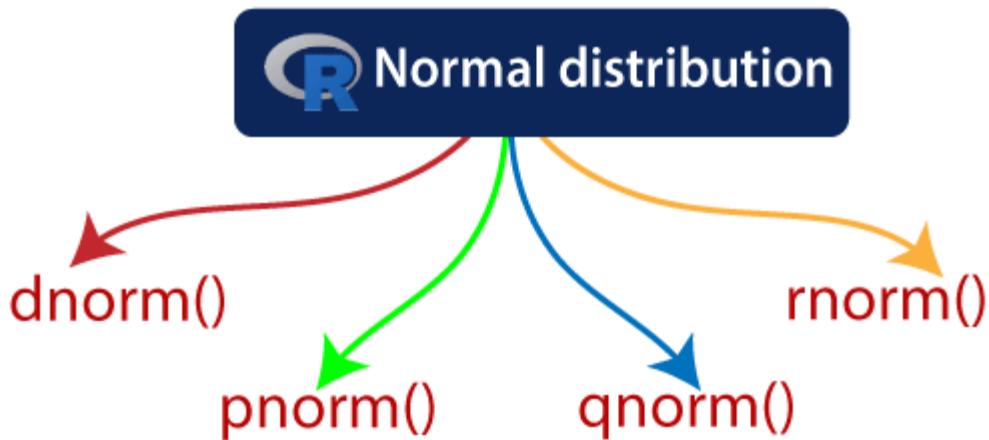
Null deviance: 297.37 on 53 degrees of freedom
Residual deviance: 210.39 on 50 degrees of freedom
AIC: 493.06

Number of Fisher Scoring iterations: 4
```

R Normal Distribution

In random collections of data from independent sources, it is commonly seen that the distribution of data is normal. It means that if we plot a graph with the value of the variable in the horizontal axis and counting the values in the vertical axis, then we get a bell shape curve. The curve center represents the mean of the data set. In the graph, fifty percent of the value is located to the left of the mean. And the other fifty percent to the right of the graph. This is referred to as the normal distribution.

R allows us to generate normal distribution by providing the following functions:



These function can have the following parameters:

S.No	Parameter	Description
1.	x	It is a vector of numbers.
2.	p	It is a vector of probabilities.
3.	n	It is a vector of observations.
4.	mean	It is the mean value of the sample data whose default value is zero.
5.	sd	It is the standard deviation whose default value is 1.

Let's start understanding how these functions are used with the help of the examples.

dnorm():Density

The dnorm() function of R calculates the height of the probability distribution at each point for a given mean and standard deviation. The probability density of the normal distribution is:

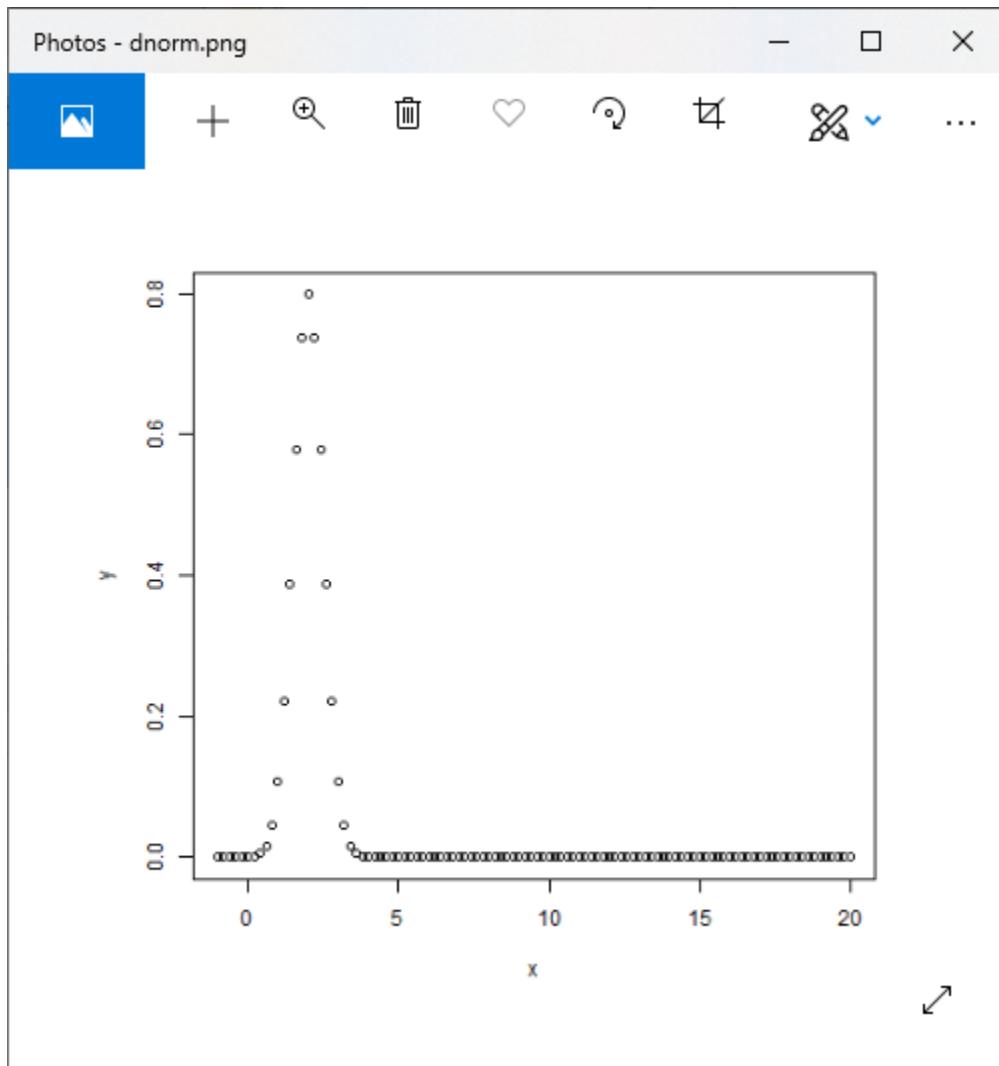
$$f(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Example

```
# Creating a sequence of numbers between -1 and 20 incrementing by 0.2.
x <- seq(-1, 20, by = .2)
# Choosing the mean as 2.0 and standard deviation as 0.5.
y <- dnorm(x, mean = 2.0, sd = 0.5)
```

```
#Plotting the graph  
plot(x,y)
```

Output:



pnorm():Direct Look-Up

The dnorm() function is also known as "Cumulative Distribution Function". This function calculates the probability of a normally distributed random numbers, which is less than the value of a given number. The cumulative distribution is as follows:

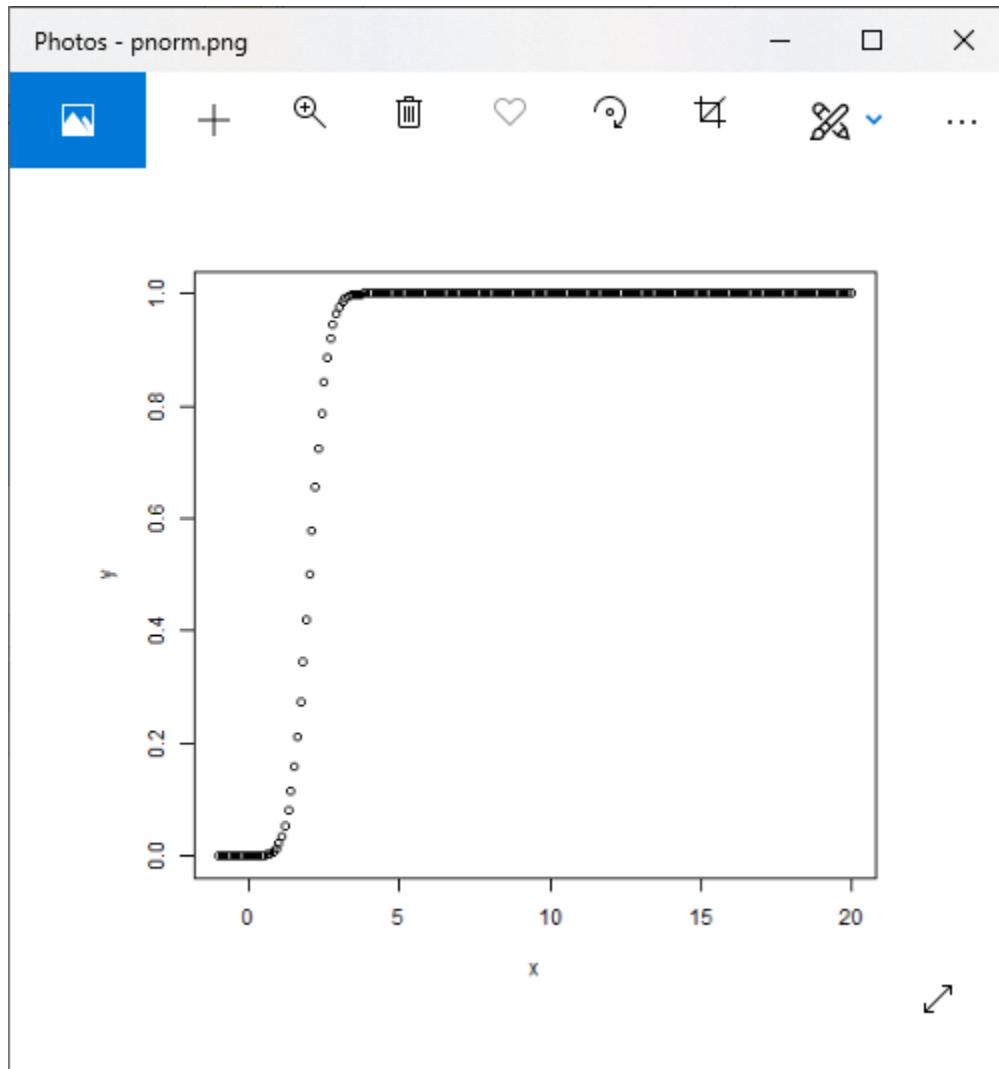
$$f(x) = P(X \leq x)$$

Example

```
# Creating a sequence of numbers between -1 and 20 incrementing by 0.2.  
x <- seq(-1, 20, by = .1)
```

```
# Choosing the mean as 2.0 and standard deviation as 0.5.
y <- pnorm(x, mean = 2.0, sd = 0.5)
#Plotting the graph
plot(x,y)
```

Output:



qnorm():Inverse Look-Up

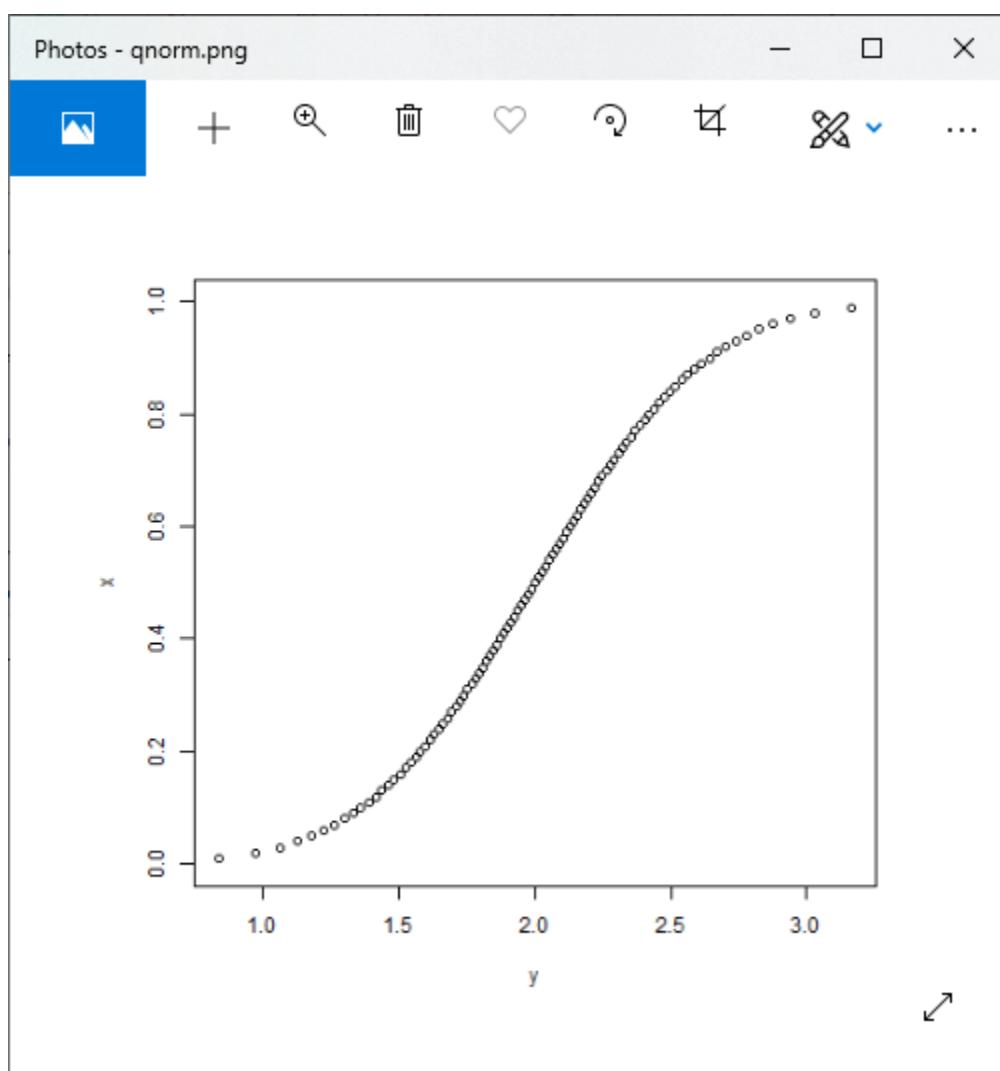
The qnorm() function takes the probability value as an input and calculates a number whose cumulative value matches with the probability value. The cumulative distribution function and the inverse cumulative distribution function are related by

$$\begin{aligned} p &= f(x) \\ x &= f^{-1}(p) \end{aligned}$$

Example

1. # Creating a sequence of numbers between -1 and 20 incrementing by 0.2.
2. x <- seq(0, 1, by = .01)
3. # Choosing the mean as 2.0 and standard deviation as 0.5.
4. y <- qnorm(x, mean = 2.0, sd = 0.5)
5. # Giving a name to the chart file.
6. png(file = "qnorm.png")
7. #Plotting the graph
8. plot(y,x)
9. # Saving the file.
10. dev.off()

Output:



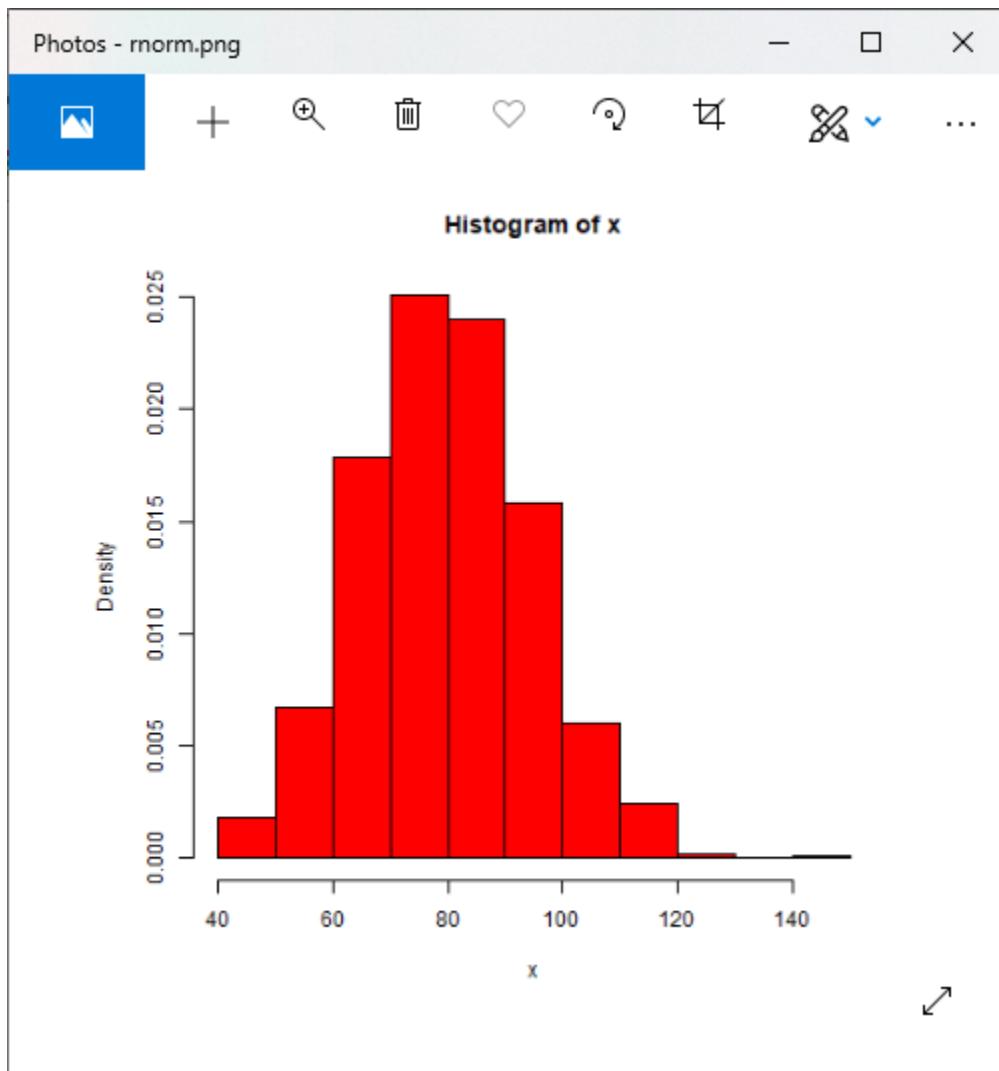
rnorm():Random variates

The rnorm() function is used for generating normally distributed random numbers. This function generates random numbers by taking the sample size as an input. Let's see an example in which we draw a histogram for showing the distribution of the generated numbers.

Example

1. # Creating a sequence of numbers between -1 and 20 incrementing by 0.2.
2. x <- rnorm(1500, mean=80, sd=15)
3. # Giving a name to the chart file.
4. png(file = "rnorm.png")
5. #Creating histogram
6. hist(x,probability = TRUE,col="red",border="black")
7. # Saving the file.
8. dev.off()

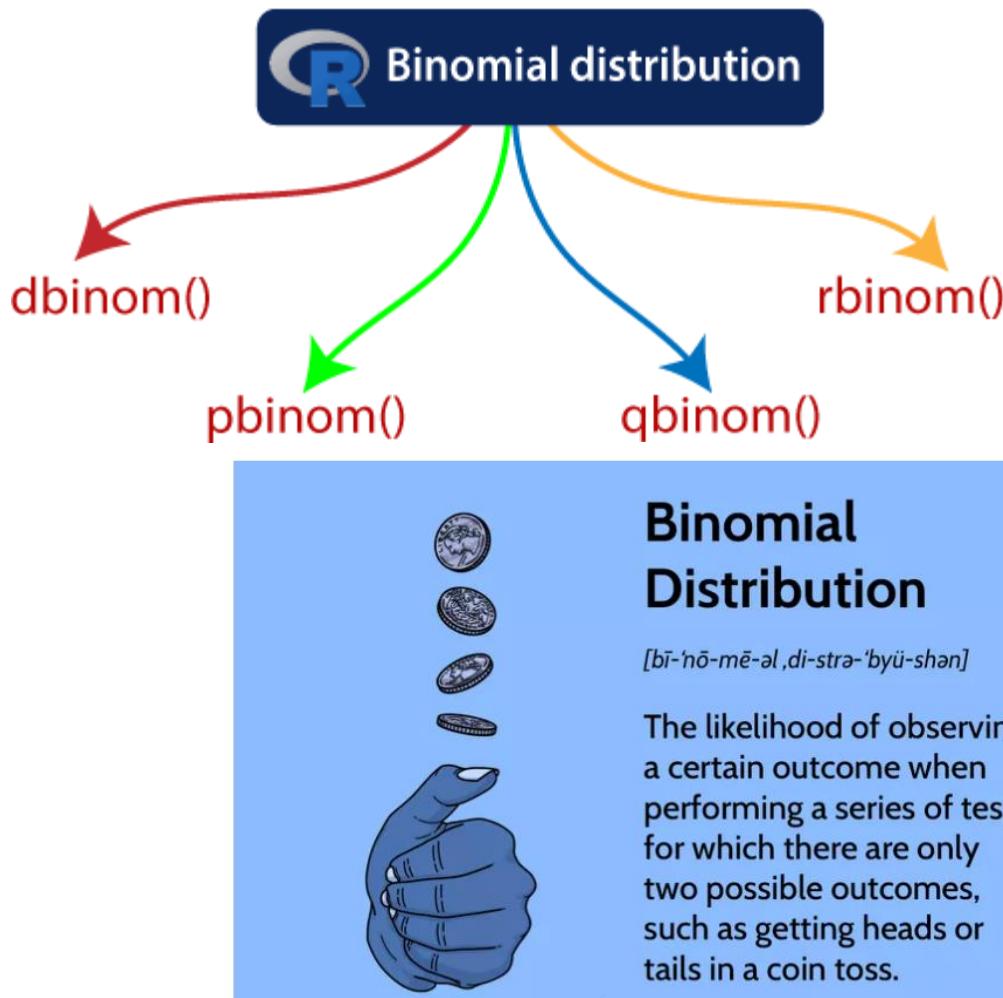
Output:



Binomial Distribution

The binomial distribution is also known as **discrete probability distribution**, which is used to find the probability of success of an event. The event has only two possible outcomes in a series of experiments. The tossing of the coin is the best example of the binomial distribution. When a coin is tossed, it gives either a head or a tail. The probability of finding exactly three heads in repeatedly tossing the coin ten times is approximate during the binomial distribution.

R allows us to create binomial distribution by providing the following function:



These function can have the following parameters:

S.No	Parameter	Description
1.	x	It is a vector of numbers.
2.	p	It is a vector of probabilities.
3.	n	It is a vector of observations.
4.	size	It is the number of trials.
5.	prob	It is the probability of the success of each trial.

Let's start understanding how these functions are used with the help of the examples

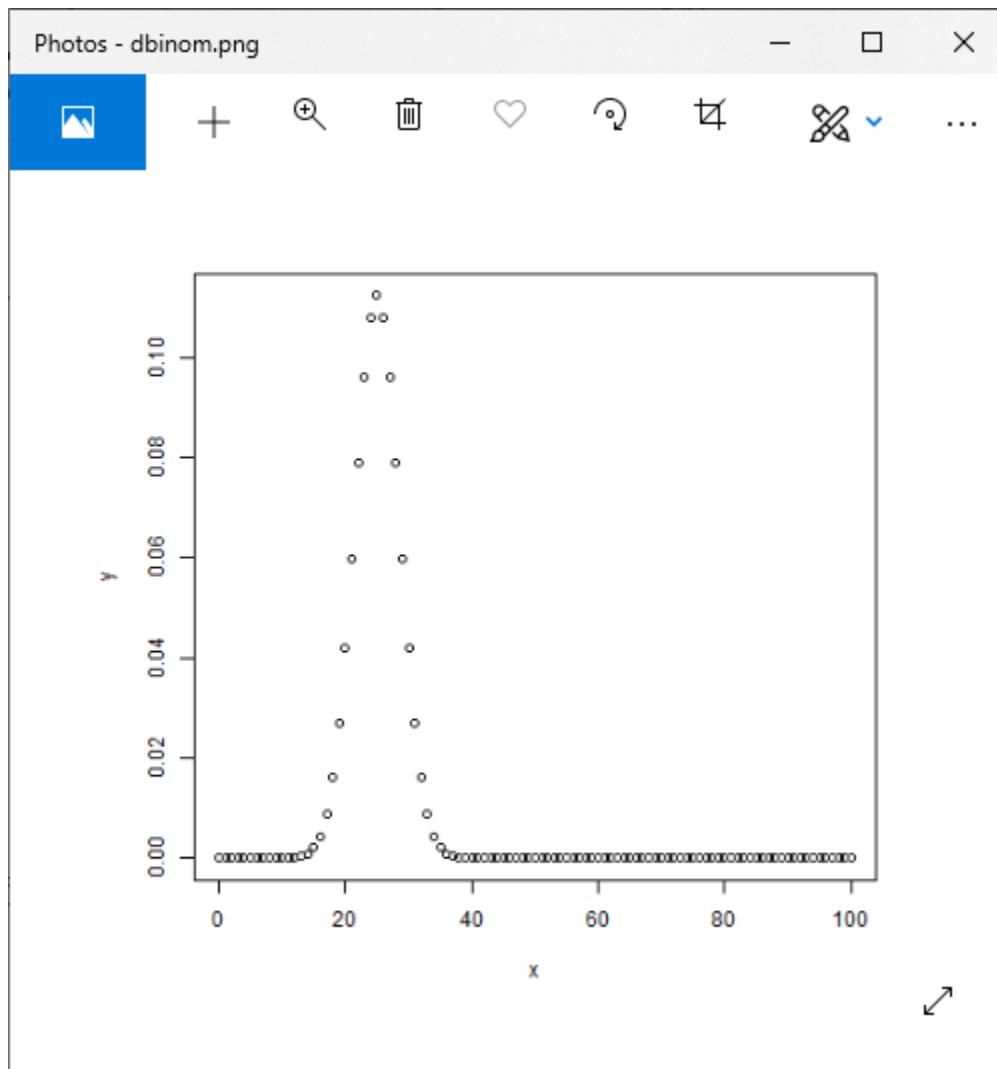
dbinom(): Direct Look-Up, Points

The `dbinom()` function of R calculates the probability density distribution at each point. In simple words, it calculates the density function of the particular binomial distribution.

Example

```
# Creating a sample of 100 numbers which are incremented by 1.5.  
x <- seq(0,100,by = 1)  
# Creating the binomial distribution.  
y <- dbinom(x,50,0.5)  
# Plotting the graph.  
plot(x,y)
```

Output:



pbinom():Direct Look-Up, Intervals

The `dbinom()` function of R calculates the cumulative probability(a single value representing the probability) of an event. In simple words, it calculates the cumulative distribution function of the particular binomial distribution.

Example

1. # Probability of getting 20 or fewer heads from 48 tosses of a coin.
2. `x <- pbinom(20,48,0.5)`
3. #Showing output
4. `print(x)`

Output:

```
> # Probability of getting 20 or less heads from a 48 tosses of a coin.
> x <- pbinom(20,48,0.5)
>
> print(x)
[1] 0.1561634
>
```

qbinom(): Inverse Look-Up

The `qbinom()` function of R takes the probability value and generates a number whose cumulative value matches with the probability value. In simple words, it calculates the inverse cumulative distribution function of the binomial distribution. Let's find the number of heads that have a probability of 0.45 when a coin is tossed 51 times.

Example

1. # Finding number of heads with the help of `qbinom()` function
2. `x <- qbinom(0.45,48,0.5)`
3. #Showing output
4. `print(x)`

Output:

```
> # Finding number of heads with the help of qbinom() function
> x <- qbinom(0.45,48,0.5)
> #Showing output
> print(x)
[1] 24
>
```

rbinom()

The rbinom() function of R is used to generate required number of random values for given probability from a given sample.

Let's see an example in which we find nine random values from a sample of 160 with a probability of 0.5.

Example

1. # Finding random values
2. x <- rbinom(9,160,0.5)
3. #Showing output
4. print(x)

Output:

```
> # Finding random values
> x <- rbinom(9,160,0.5)
> #Showing output
> print(x)
[1] 77 84 88 76 79 65 89 79 82
> |
```

R Classification

The idea of the classification algorithm is very simple. We predict the target class by analyzing the training dataset. We use training datasets to obtain better boundary conditions that can be used to determine each target class. Once the boundary condition is determined, the next task is to predict the target class. The entire process is known as classification.

There are some important points of classification algorithms:



- **Classifier**
It is an algorithm that maps the input data to a specific category.
- **Classification Model**
A classification model tries to draw some conclusions from the input values which are given for training. This conclusion will predict class labels/categories for new data.
- **Feature**
It is an individual measurable property of an event being observed.
- **Binary classification**
It is a classification task that has two possible outcomes. E.g., Gender classification, which has only two possible outcomes, i.e., Male and Female.
- **Multi-class classification**
It is a classification task in which classification is done with more than two classes. An example of multi-class classification is: an animal can be a dog or cat, but not both at the same time.
- **Multi-label classification**
It is a classification task in which each sample is mapped with a set of target

labels. An example of multi-label classification is: a news article that can be about a person, location, and sports at the same time.

Types of Classification Algorithms

In R, classification algorithms are broadly classified in the following types:

- **Linear classifier**

In machine learning, the main task of statistical classification is to use an object's characteristics for finding to which class it belongs. This task is achieved by making a classification decision based on the value of a linear combination of the characteristics. In R, there are three linear classification algorithms which are as follows:

1. Logistic Regression
2. Naive Bayes classifier
3. Fisher's linear discriminant

- **Support vector machines**

A support vector machine is the supervised learning algorithm that analyzes data that are used for classification and regression analysis. In SVM, each data item is plotted as a point in n-dimensional space with the value of each attribute, that is the value of a particular coordinate.

Least squares support vector machines is mostly used classification algorithm in R.

- **Quadratic classifiers**

Quadratic classification algorithms are based on Bayes theorem. These classifiers algorithms are different in their approach for classification from the logistic regression. In logistic regression, it is possible to derive the probability of observation directly for a class ($Y = k$) for a particular observation ($X = x$). But in quadratic classifiers, the observation is done in the following two steps:

1. In the first step, we identify the distribution for input X for each of the groups or classes.
2. After that, we flip the distribution with the help of Bayes theorem to calculate the probability.

- **Kernel estimation**

Kernel estimation is a non-parametric way of estimating the **Probability Density Function** (PDF) of the continuous random variable. It is non-

parametric because it assumes no implicit distribution for the variable. Essentially, on each datum, a kernel function is created with the datum at its center. It ensures that the kernel is symmetric about the datum. The PDF is then estimated by adding all these kernel functions and dividing it by the number of data to ensure that it satisfies the two properties of the PDF:

1. Every possible value of the PDF should be non-negative.
2. The fixed integral of the PDF on its support set should be equal to 1.

In R, the k-nearest neighbor is the most used kernel estimation algorithm for classification.

- **Decision Trees**

Decision Tree is a supervised learning algorithm that is used for classification and regression tasks. In R, the decision tree classifier is implemented with the help of the R machine learning caret package. The random forest algorithm is the mostly used decision tree algorithm used in R.

- **Neural Networks**

The neural network is another classifier algorithm that is inspired by the human brain for performing a particular task or function. These algorithms are mostly used in image classification in R. To implement neural network algorithms, we have to install the **neuralnet** package.

- **Learning vector quantization**

Learning vector quantization is a classification algorithm that is used for binary and multi-class problems. By learning the training dataset, the LVQ model creates codebook vectors that represent class regions. They contain elements which are placed around the respective class according to their matching level. If the element matches, it moves closer to the target class, if it does not match, then it proceeds.

R-Time Series Analysis

Any metric which is measured over regular time intervals creates a time series. Analysis of time series is commercially important due to industrial necessity and relevance, especially with respect to the forecasting (demand, supply, and sale, etc.). A series of data points in which each data point is associated with a timestamp is known as time series.

The stock price at different points in a day in the stock market is the simplest example of the time series. The amount of rainfall in an area in different months of the year is another example of it. R provides several functions for creating, manipulating, and plotting time series data. In the R-object, the time series data is known as the time-series object. It is just like a vector or data frame.

Creating a Time Series

R provides `ts()` function for creating a Time Series. There is the following syntax of the `ts()` function:

1. `Timeseries_object_name <- ts(data, start, end, frequency)`

Here,

S.No	Parameter	Description
1.	<code>data</code>	It is a vector or matrix which contains the value used in time series.
2.	<code>start</code>	It is the start time for the first observation
3.	<code>end</code>	It is the end time for the last observation
4.	<code>frequency</code>	It specifies the number of observations per unit time.

Let's see an example to understand how `ts()` function is used for creating Time Series.

Example:

In the below example, we will consider the annual snowfall details at a place starting from January 2013. We will create an R time series object for a period of 12 month and plot it.

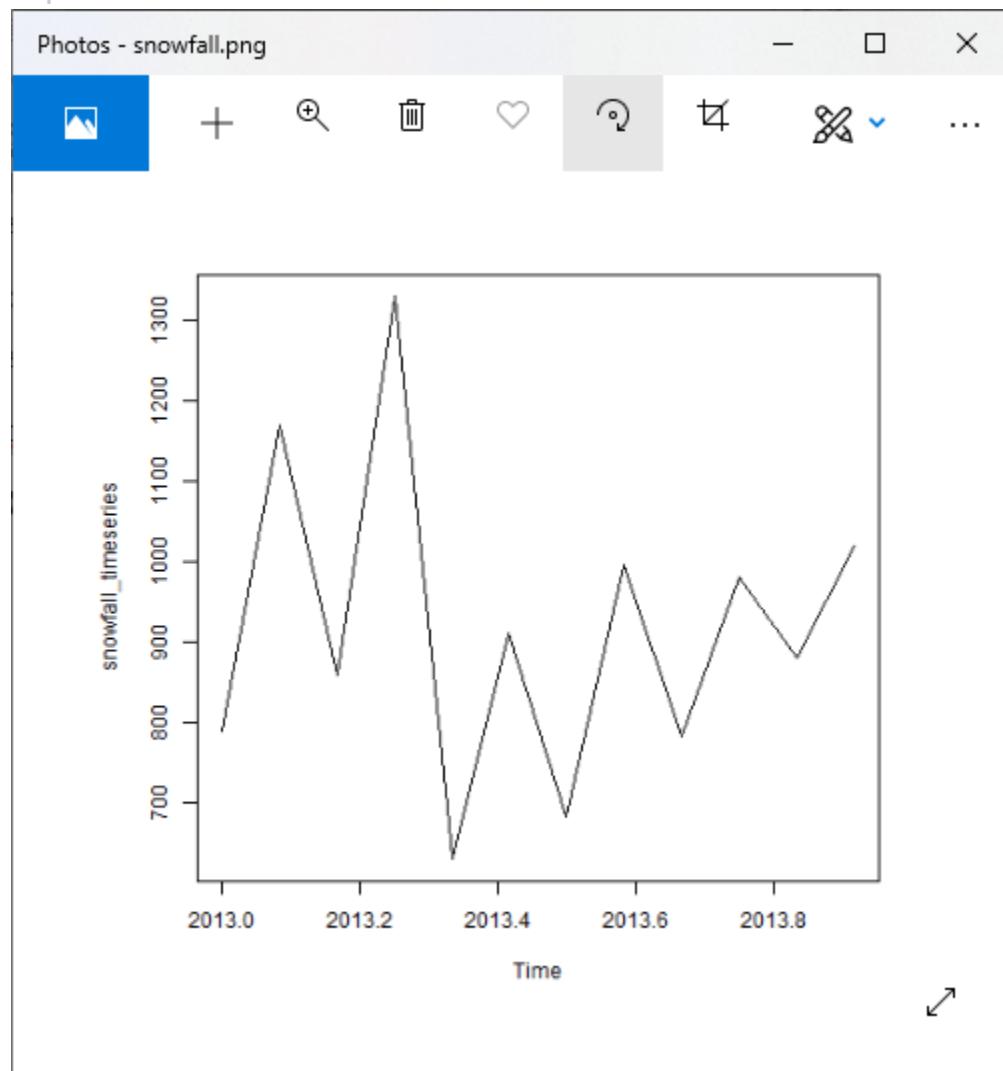
```
# Getting the data points in form of a R vector.  
snowfall <- c(790,1170.8,860.1,1330.6,630.4,911.5,683.5,996.6,783.2,982,881.8,1021)  
# Convertting it into a time series object.  
snowfall_timeseries<- ts(snowfall,start = c(2013,1),frequency = 12)  
# Printing the timeseries data.  
print(snowfall_timeseries)
```

```
# Plotting a graph of the time series.
```

```
plot(snowfall_timeseries)
```

Output:

```
> # Getting the data points in form of a R vector.  
> snowfall <- c(790,1170.8,860.1,1330.6,630.4,911.5,683.5,996.6,783.2,982,881.8,1021)  
>  
> # Convertting it into a time series object.  
> snowfall_timeseries <- ts(snowfall,start = c(2013,1),frequency = 12)  
>  
> # Printing the timeseries data.  
> print(snowfall_timeseries)  
      Jan    Feb    Mar    Apr    May    Jun    Jul    Aug    Sep    Oct    Nov    Dec  
2013  790.0 1170.8  860.1 1330.6  630.4  911.5  683.5  996.6  783.2  982.0  881.8 1021.0  
>  
> # Giving a name to the chart file.  
> png(file = "snowfall.png")  
>  
> # Plotting a graph of the time series.  
> plot(snowfall_timeseries)  
>  
> # Saving the file.  
> dev.off()  
null device  
1
```



What is Stationary Time Series?

A Stationary Time Series is a time series, if:

1. The mean value of the time-series is constant over time. This implies that the trend component is declared null.
2. The variance should not increase over time.
3. The seasonality effect should be minimal.

This means that it is devoid of or trendseasonal patterns, which resemble a random white noise regardless of the time interval observed.

In simple words, a stationary time series is the one whose statistical properties like mean, variance, and autocorrelation, etc. are all constant over time.

Extracting the trend, seasonality, and error

We can decompose the time series by splitting it into three components, such as seasonality, trends, and random fluctuations.

Time series decomposition is a mathematical process that transforms a time series into multiple time series.

Seasonal:

Patterns that repeat over a certain period of time

Trend:

An underlying trend of the matrices.

Random:

It is the residuals of the original time series after the seasonal and trend series are removed.

Additive and Multiplicative Decomposition

Additive and Multiplicative decompositions are the models that are used for analyzing the series. When the seasonal variation seems to be constant means when the seasonal variation does not change at the time when the value of the time series increases, then we use the Additive model else we use the multiplicative model.

Decomposition Model

Additive:

Time series = Seasonal + Trend + Random

Multiplicative:

Time series = Trend * Seasonal *Random

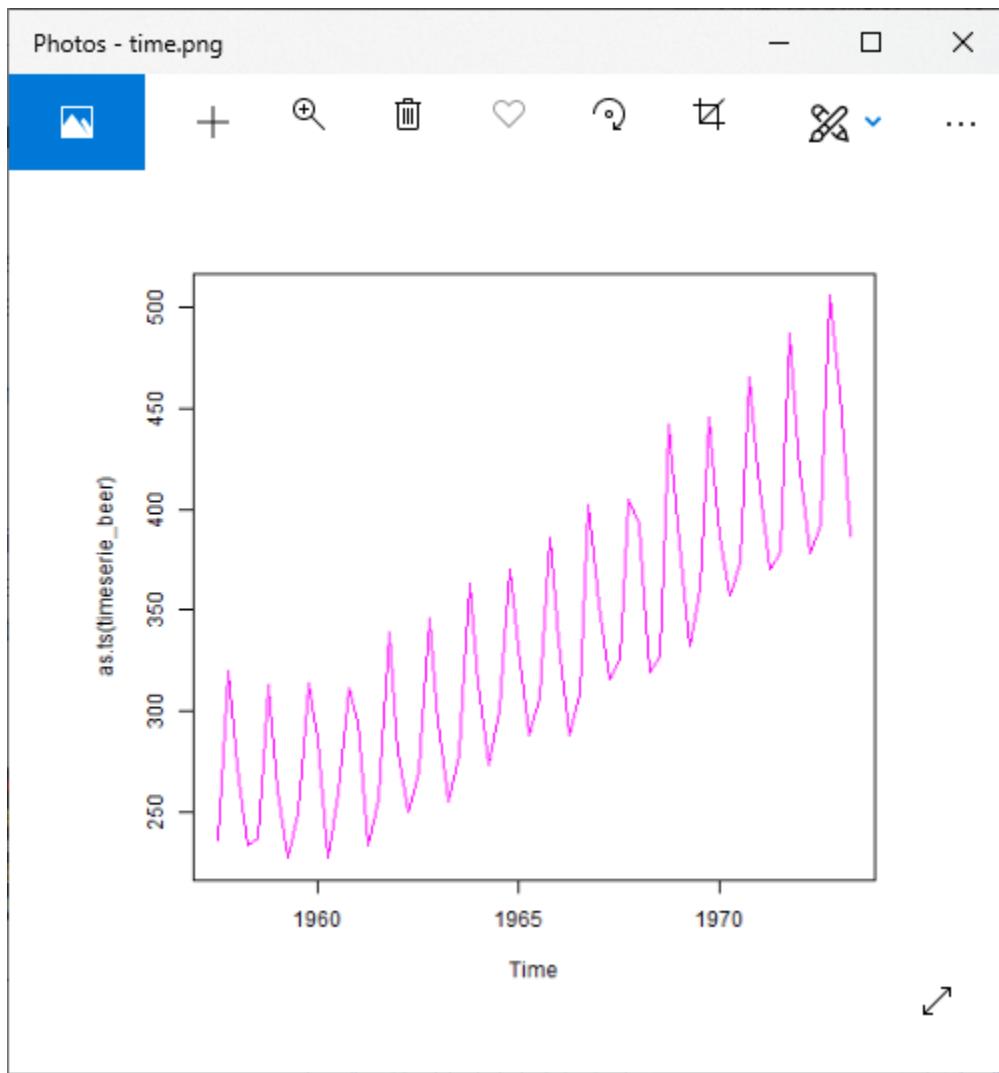
Let's see a step by step procedure to understand how we can decompose the time series using both Additive and Multiplicative model. For the additive model, we use ausbeer dataset, and for multiplicative, we use the AirPassengersdataset.

Step 1: Load data and create time series

For Additive Model

```
#Importing library fpp
library(fpp)
#Using ausbeer data
data(ausbeer)
#Creating time series for ausbeer dataset
timeserie.beer = tail(head(ausbeer, 17*4+2),17*4-4)
plot(as.ts(timeserie_beer), col="magenta")
```

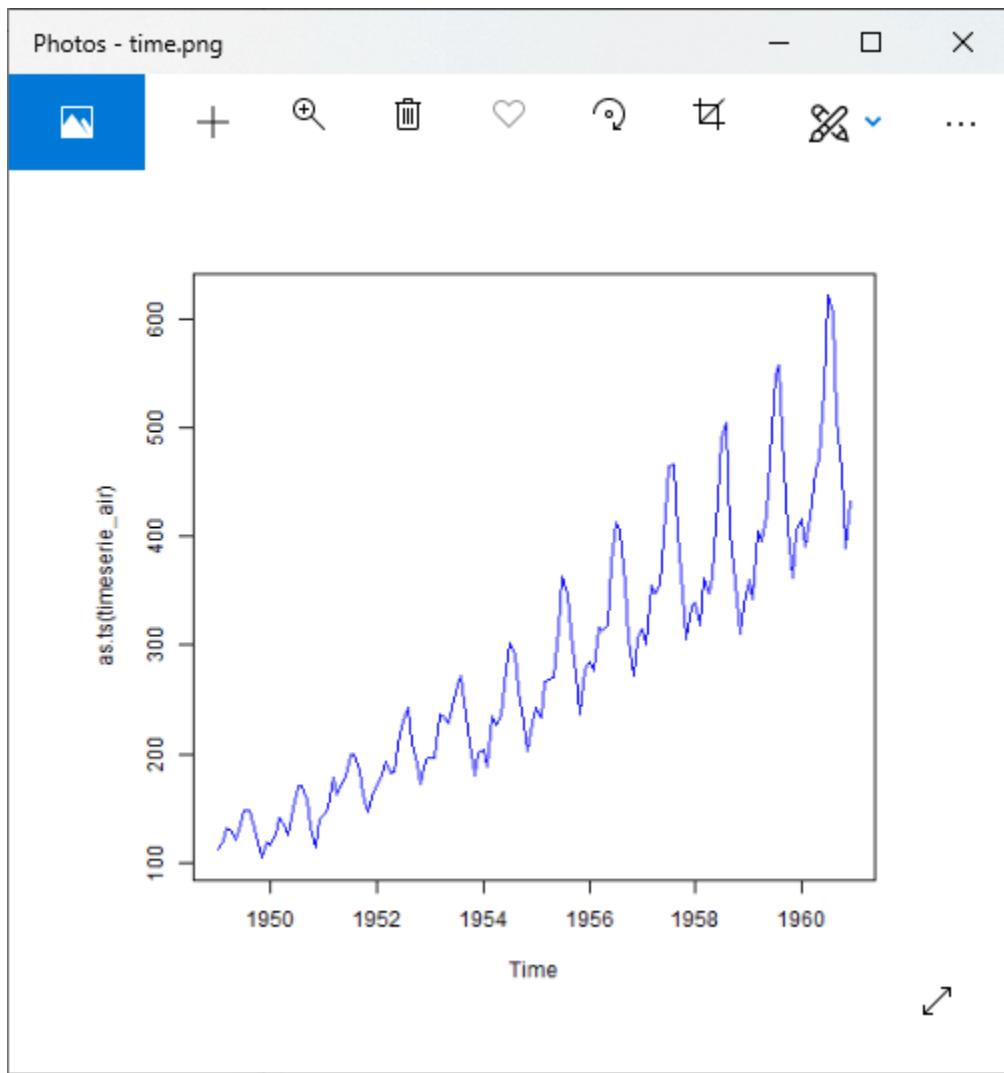
Output:



For Multiplicative Model

```
#Importing library Ecdat
library(Ecdat)
#Using AirPassengers data
data(AirPassengers)
#Creating time series for AirPassengers dataset
timeserie_air = AirPassengers
# Giving a name to the file.
png(file = "time.png")
plot(as.ts(timeserie_air))
# Saving the file.
dev.off()
```

Output:

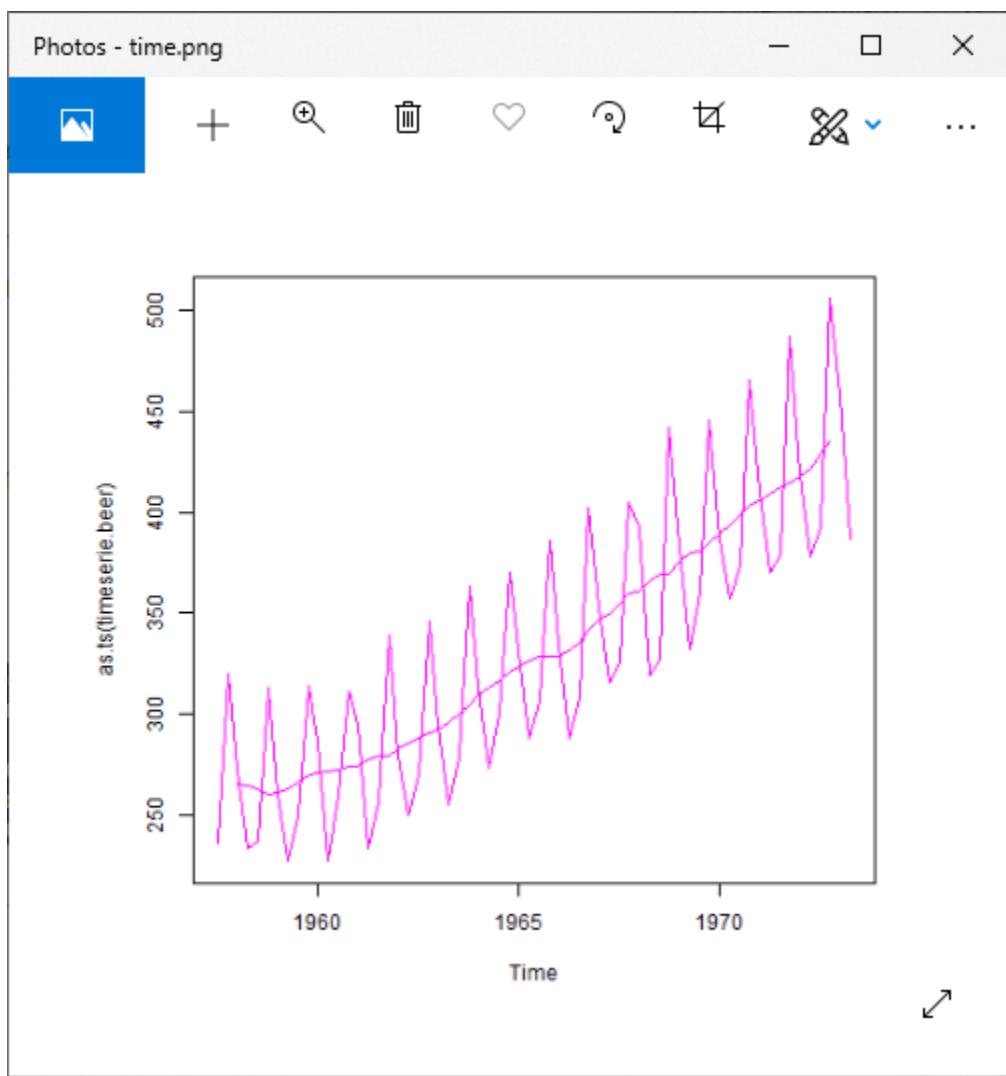


Step 2: Detecting the Trend

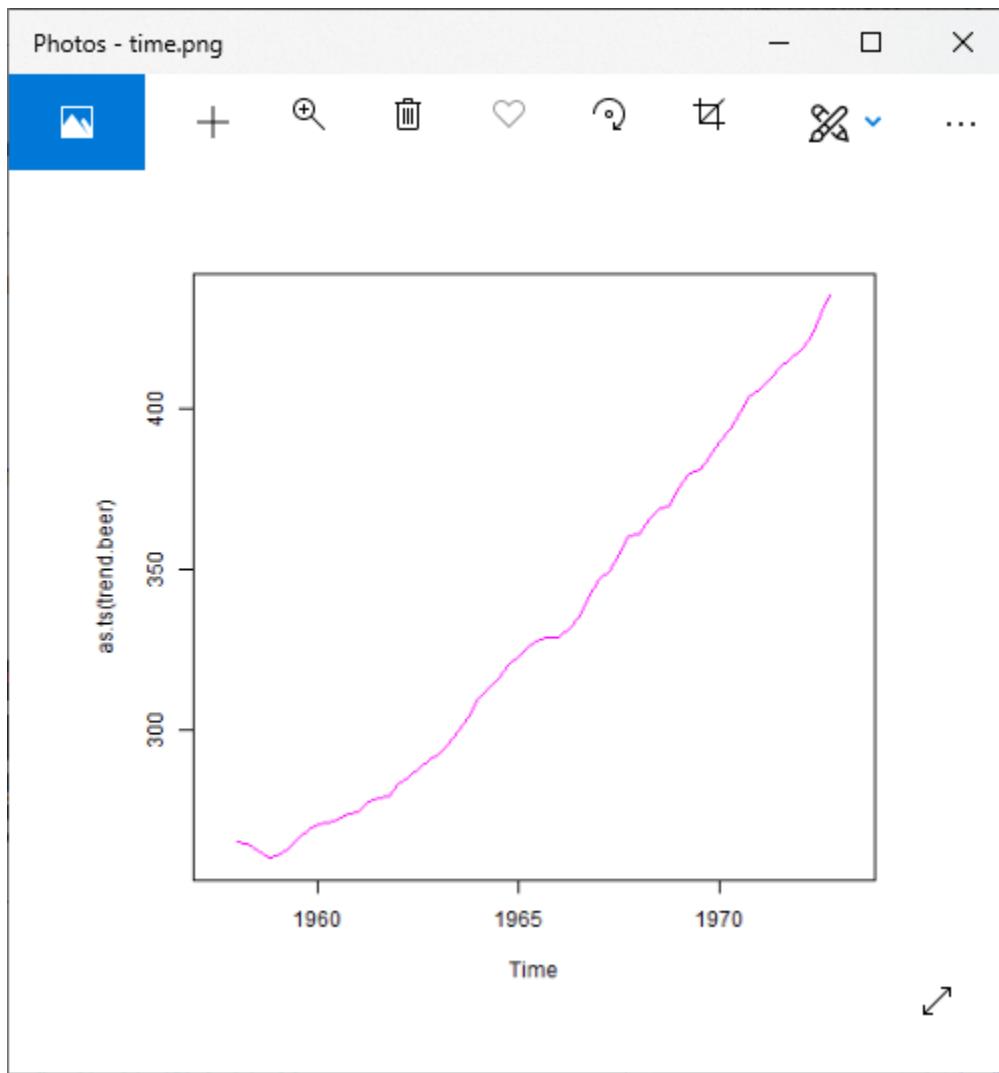
For Additive Model

1. #Detecting trend
2. `trend.beer = ma(timeserie.beer, order = 4, centre = T)`
3. # Giving a name to the file.
4. `png(file = "time.png")`
5. `plot(as.ts(timeserie.beer), col="red")`
6. `lines(trend.beer,col="red")`
7. `plot(as.ts(trend.beer),col="red")`
8. # Saving the file.
9. `dev.off()`

Output1:



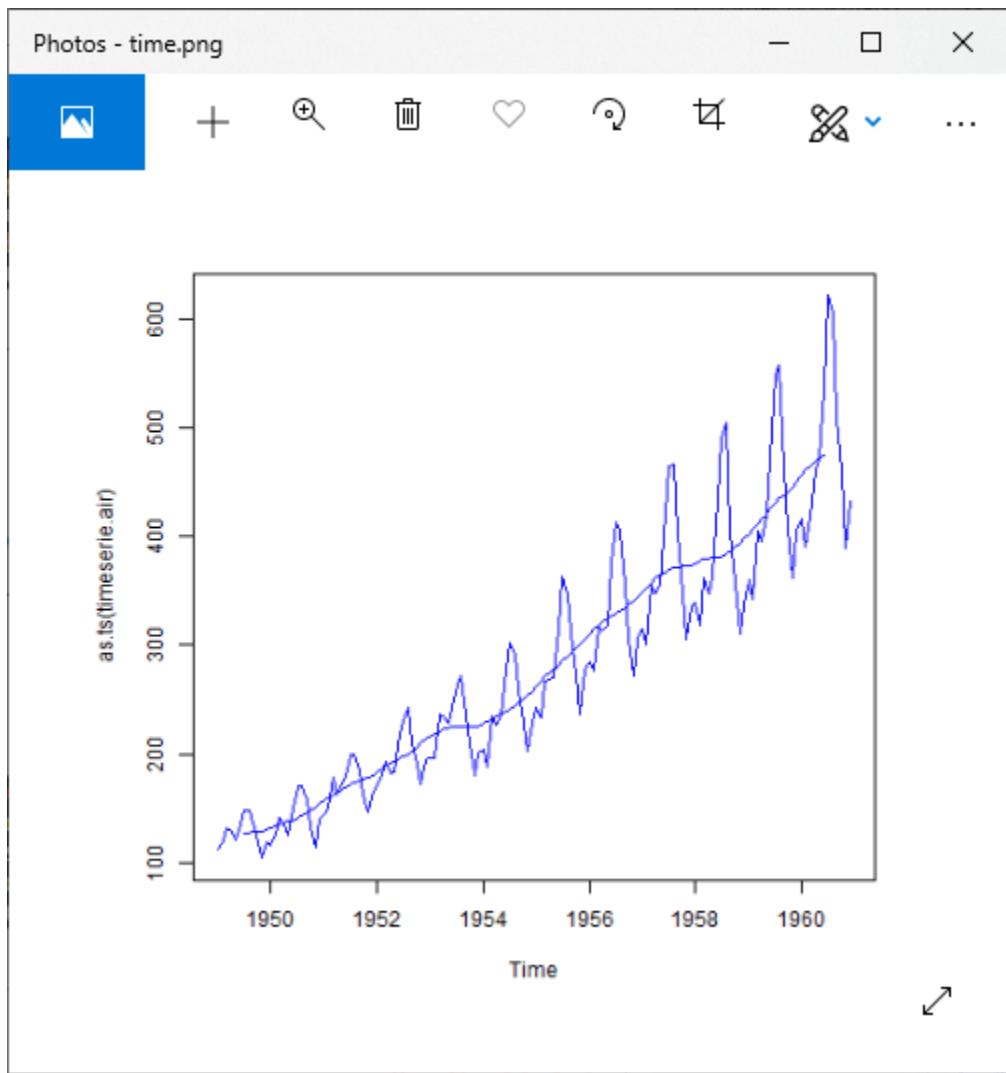
Output2:



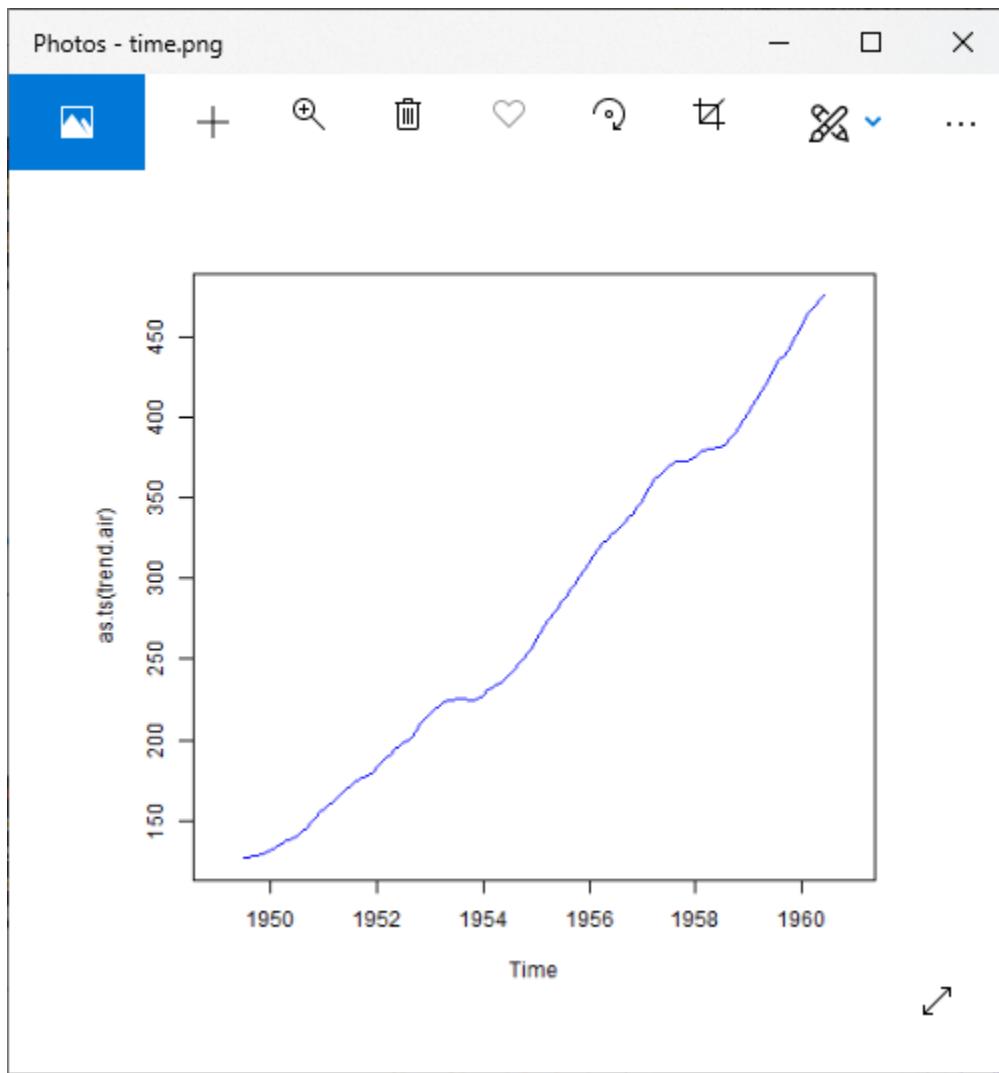
For Multiplicative Model:

1. #Detecting trend
2. `trend.air = ma(timeserie.air, order = 12, centre = T)`
3. # Giving a name to the file.
4. `png(file = "time.png")`
5. `plot(as.ts(timeserie.air), col = "blue")`
6. `lines(trend.air, col = "blue")`
7. `plot(as.ts(trend.air), col = "blue")`
8. # Saving the file.
9. `dev.off()`

Output1:



Output2:

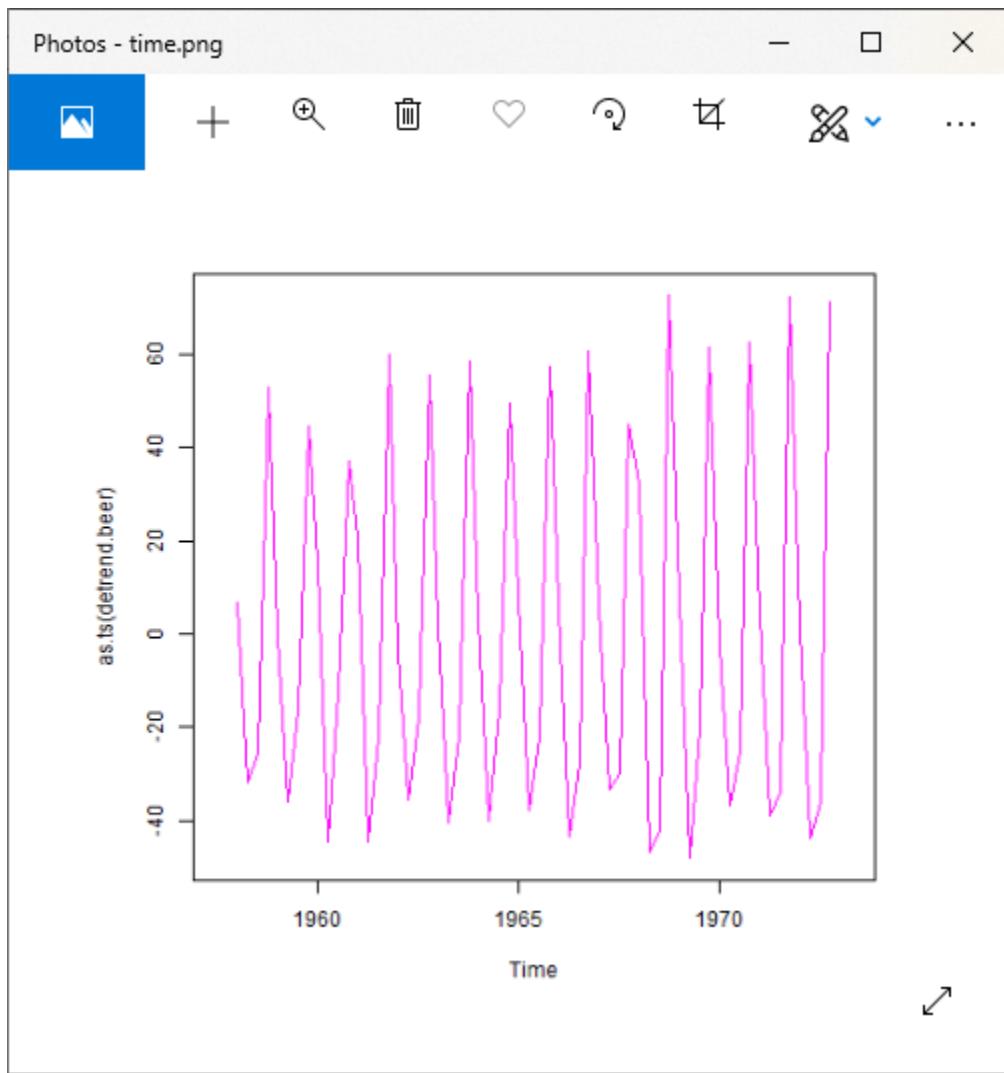


Step 3: Detrend of Time Series

For Additive Model

1. #Detrend the time series.
2. `detrend.beer=timeserie.beer-trend.beer`
3. # Giving a name to the file.
4. `png(file = "time.png")`
5. `plot(as.ts(detrend.beer),col="magenta")`
6. # Saving the file.
7. `dev.off()`

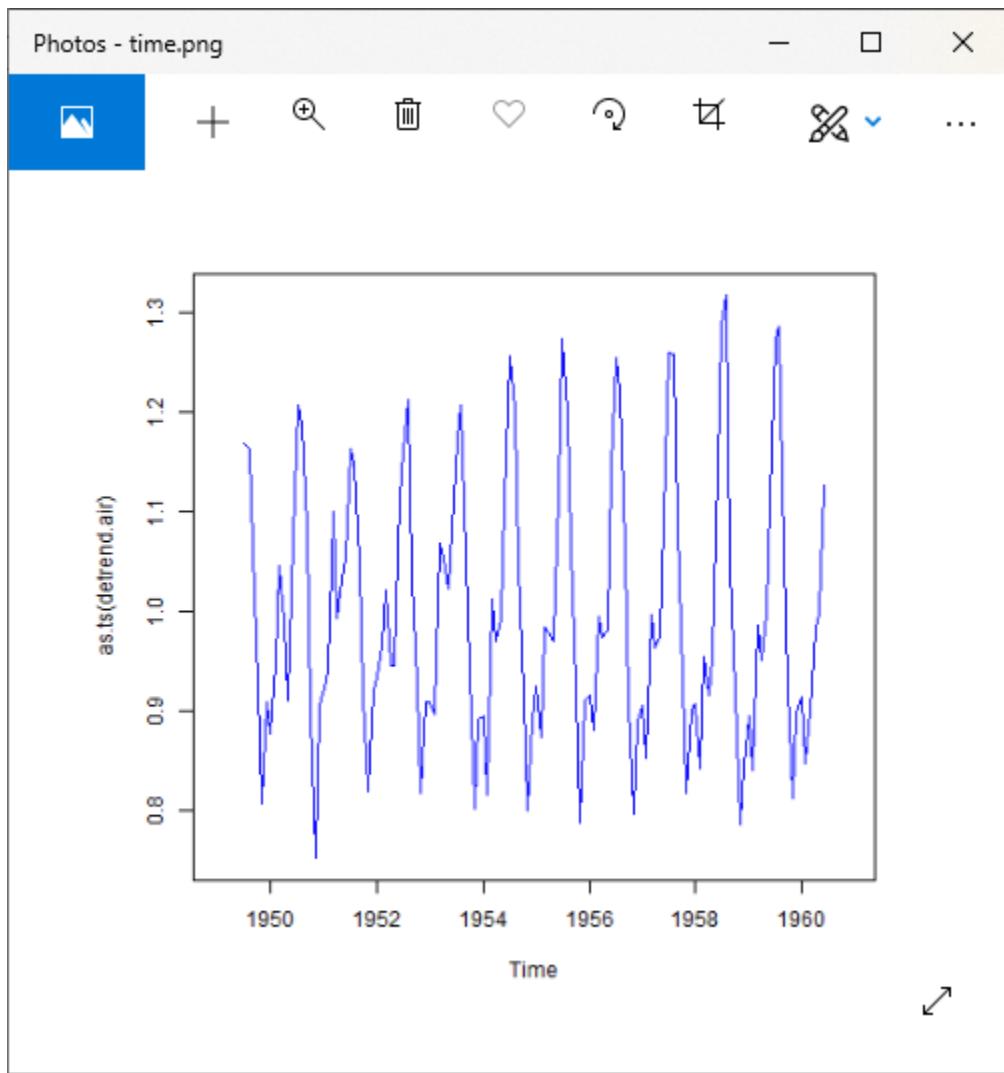
Output:



For Multiplicative Model

1. #Detrend of time series
2. `detrend.air=timeserie.air / trend.air`
3. # Giving a name to the file.
4. `png(file = "time.png")`
5. `plot(as.ts(detrend.air), col="blue")`
6. # Saving the file.
7. `dev.off()`

Output:

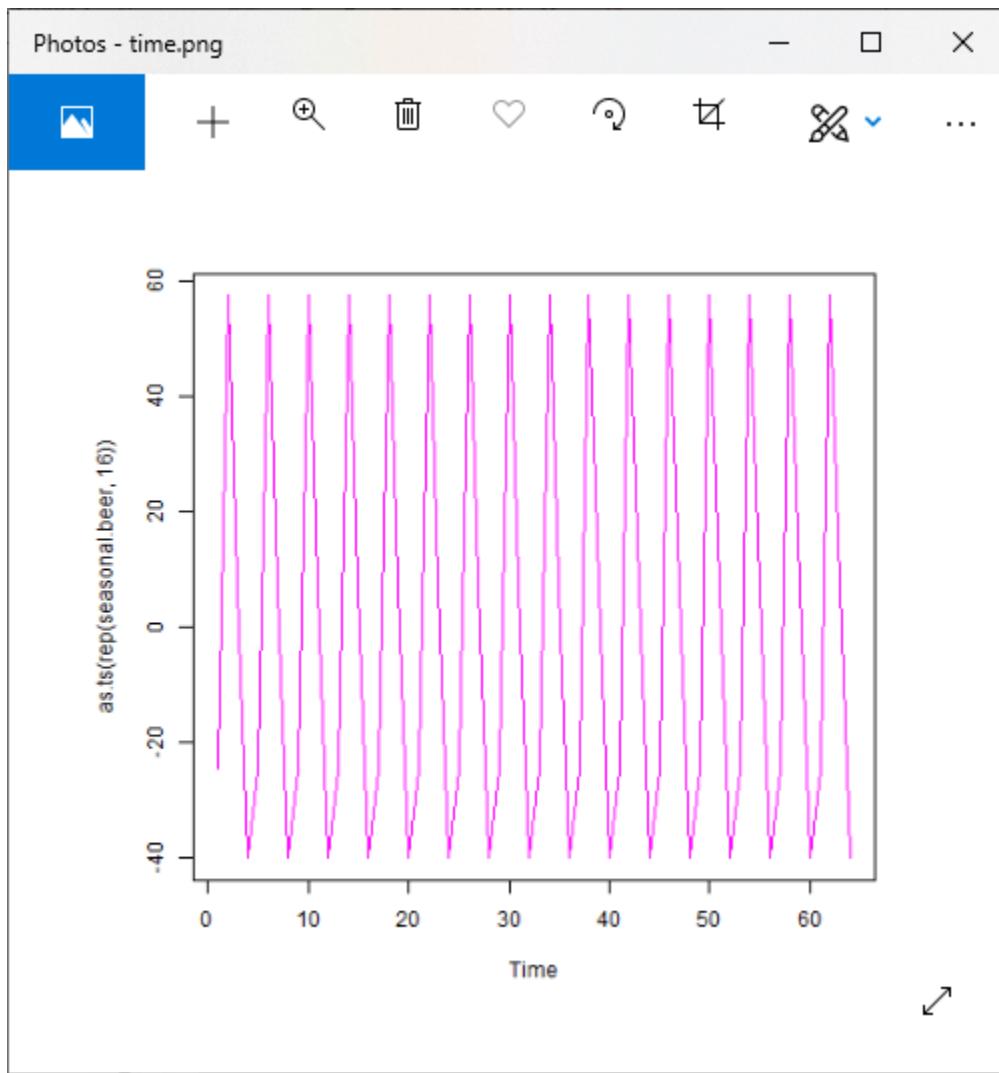


Step 4: Average the Seasonality

For Additive Model

1. #Average the seasonality
2. `m.beer = t(matrix(data = detrend.beer, nrow = 4))`
3. `seasonal.beer = colMeans(m.beer, na.rm = T)`
4. # Giving a name to the file.
5. `png(file = "time.png")`
6. `plot(as.ts(rep(seasonal.beer,16)),col="magenta")`
7. # Saving the file.
8. `dev.off()`

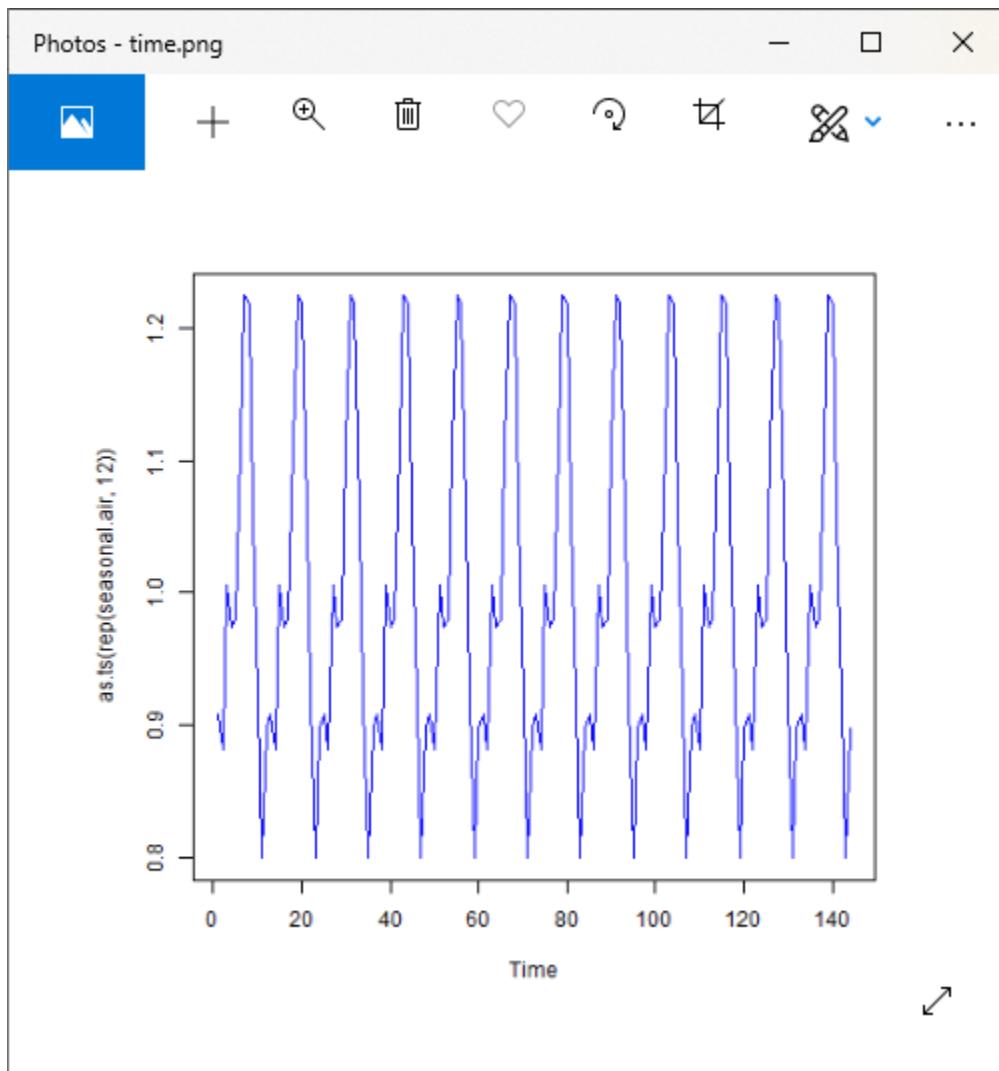
Output:



For Multiplicative Model

1. #Average the seasonality
2. `m.air = t(matrix(data = detrend.air, nrow = 12))`
3. `seasonal.air = colMeans(m.air, na.rm = T)`
4. # Giving a name to the file.
5. `png(file = "time.png")`
6. `plot(as.ts(rep(seasonal.air,12)),col="blue")`
7. # Saving the file.
8. `dev.off()`

Output:

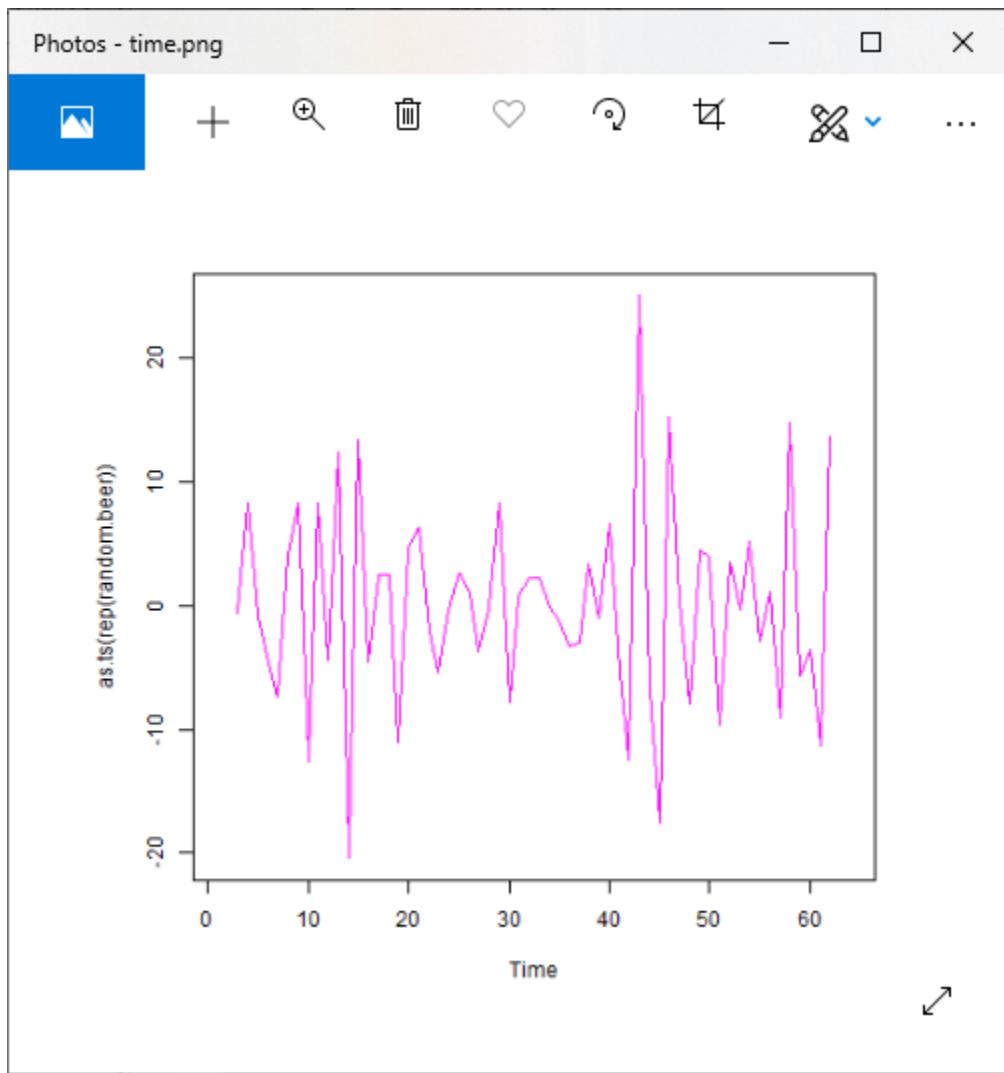


Step 5: Examining the Remaining Random Noise

For Additive Model

1. # Examining the Remaining Random Noise
2. `random.beer = timeserie.beer - trend.beer - seasonal.beer`
3. # Giving a name to the file.
4. `png(file = "time.png")`
5. `plot(as.ts(rep(random.beer)), col="magenta")`
6. # Saving the file.
7. `dev.off()`

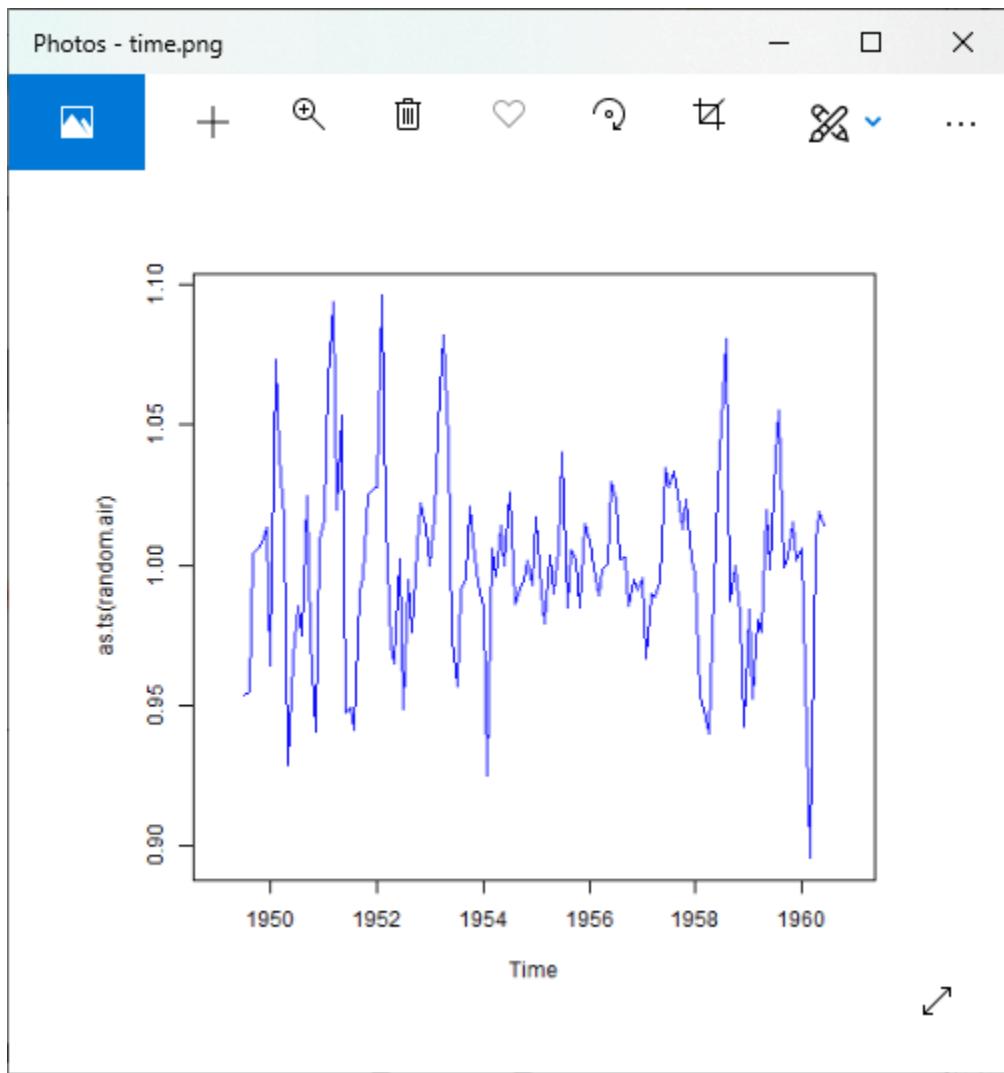
Output:



For Multiplicative Model

1. # Examining the Remaining Random Noise
2. `random.air = timeserie.air / (trend.air * seasonal.air)`
3. # Giving a name to the file.
4. `png(file = "time.png")`
5. `plot(as.ts(random.air), col="blue")`
6. # Saving the file.
7. `dev.off()`

Output:

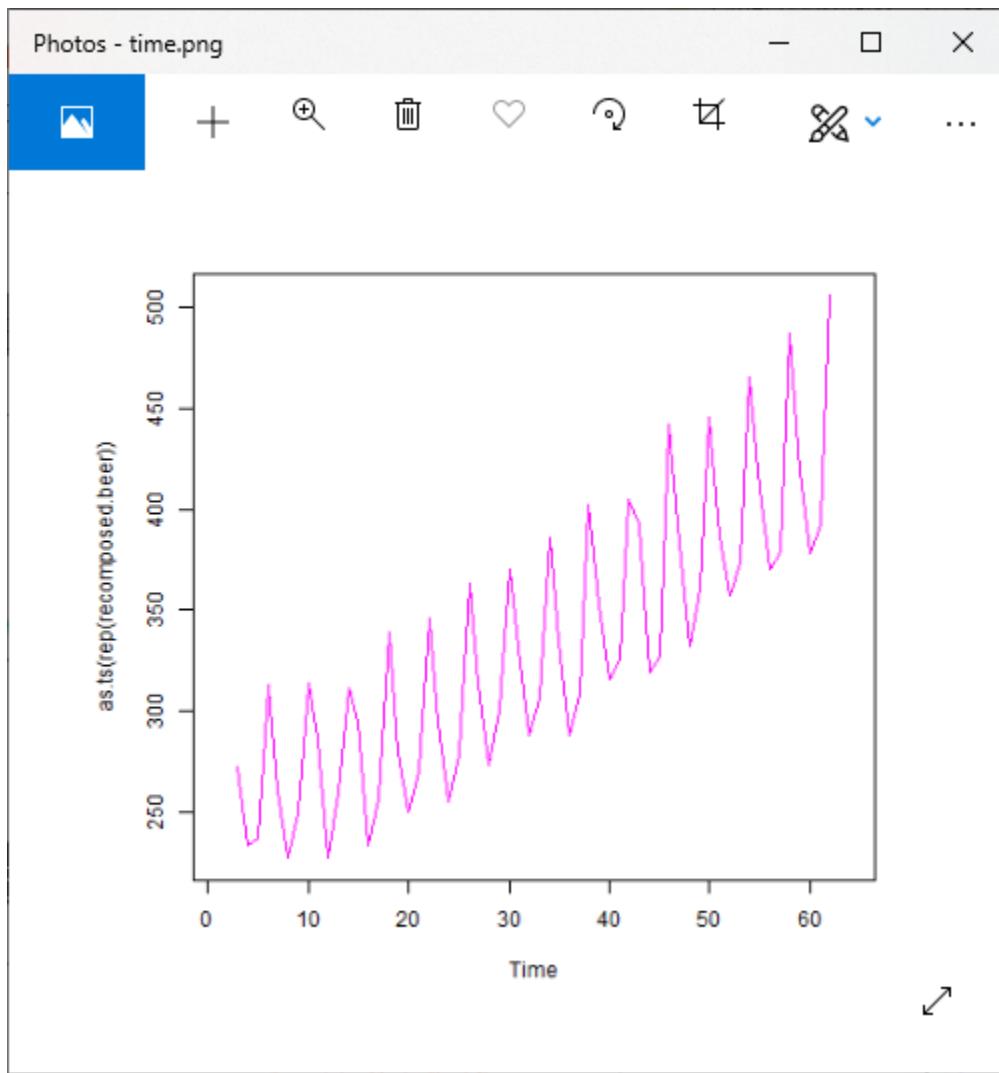


Step 5: Reconstruction of Original Signal

For Additive Model

1. #Reconstruction of original signal
2. recomposed.beer=trend.beer+seasonal.beer+random.beer
3. # Giving a name to the file.
4. png(file = "time.png")
5. plot(as.ts(recomposed.beer),col="magenta")
6. # Saving the file.
7. dev.off()

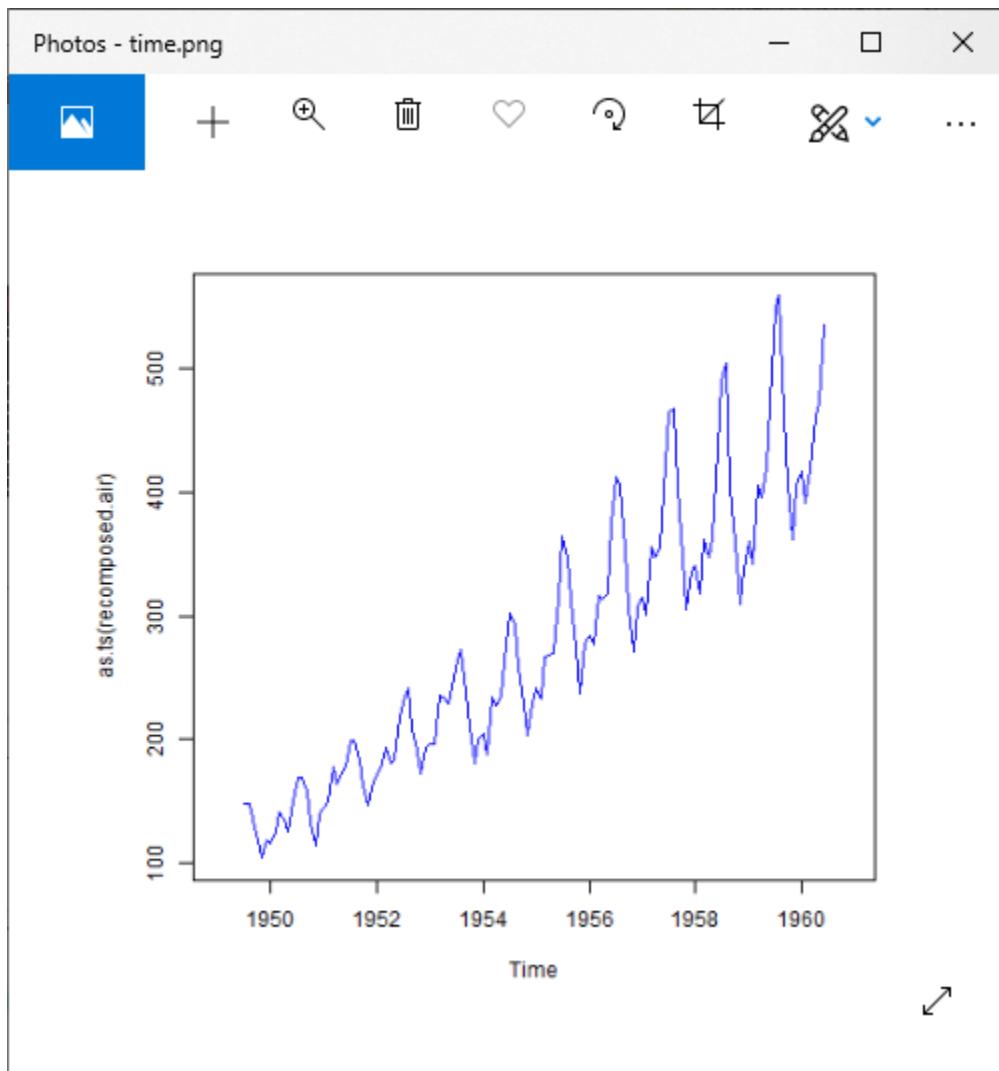
Output1:



For Multiplicative Model

1. #Reconstruction of original signal
2. `recomposed.air = trend.air*seasonal.air*random.air`
3. # Giving a name to the file.
4. `png(file = "time.png")`
5. `plot(as.ts(recomposed.air), col="blue")`
6. # Saving the file.
7. `dev.off()`

Output:



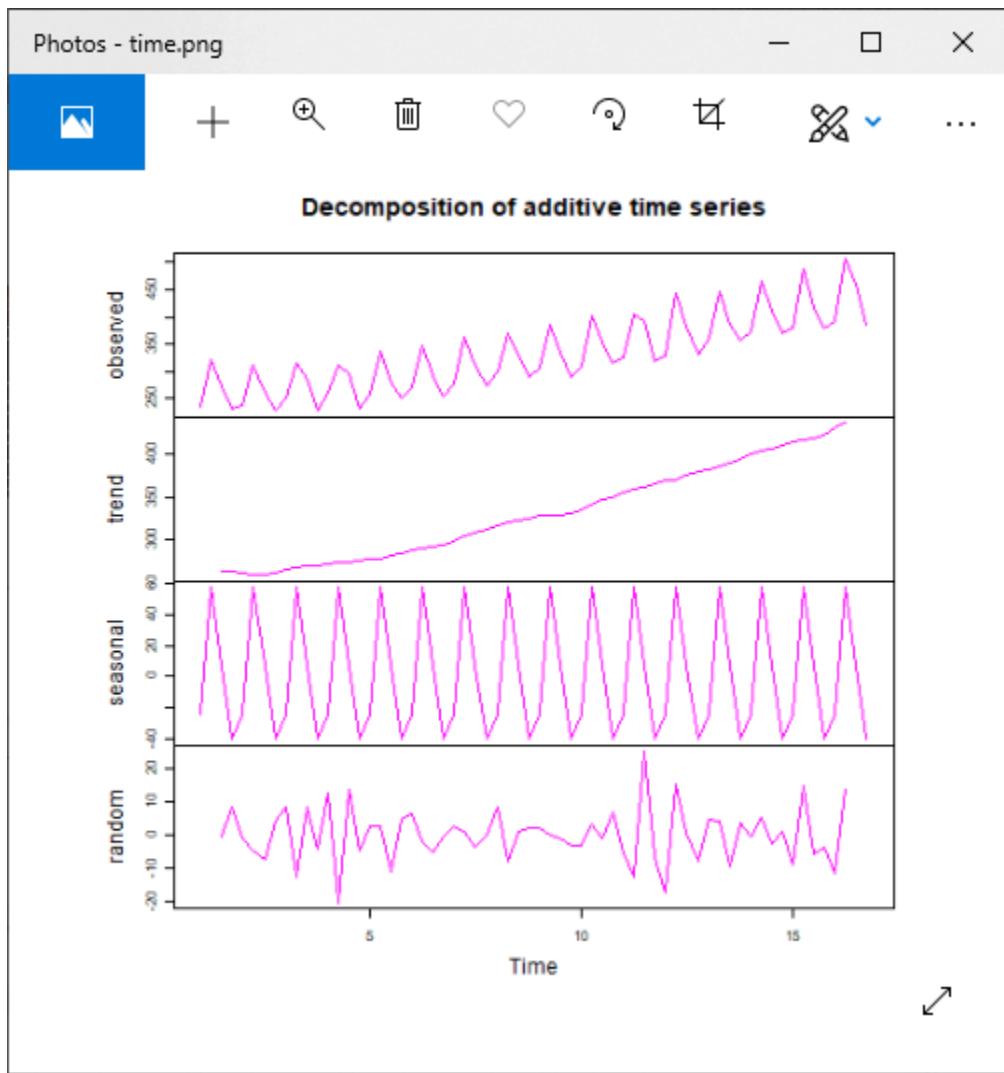
Time Series Decomposition using decompose()

For Additive Model

1. #Importing libraries
2. library(forecast)
3. library(timeSeries)
4. library(fpp)
5. #Using ausbeer data
6. data(ausbeer)
7. #Creating time series
8. **timeserie.beer = tail(head(ausbeer, 17*4+2),17*4-4)**
9. #Detect trend
10. **trend.beer = ma(timeserie.beer, order = 4, centre = T)**
11. #Detrend of time series

```
12. detrend.beer=timeserie.beer-trend.beer
13. #Average the seasonality
14. m.beer = t(matrix(data = detrend.beer, nrow = 4))
15. seasonal.beer = colMeans(m.beer, na.rm = T)
16. #Examine the remaining random noise
17. random.beer = timeserie.beer - trend.beer - seasonal.beer
18. #Reconstruct the original signal
19. recomposed.beer = trend.beer+seasonal.beer+random.beer
20. #Decomposed the time series
21. ts.beer = ts(timeserie.beer, frequency = 4)
22. decompose=decompose.beer = decompose(ts.beer, "additive")
23. # Giving a name to the file.
24. png(file = "time.png")
25. par(mfrow=c(2,2))
26. plot(as.ts(decompose.beer$seasonal),col="magenta")
27. plot(as.ts(decompose.beer$trend),col="magenta")
28. plot(as.ts(decompose.beer$random),col="magenta")
29. plot(decompose.beer,col="magenta")
30. # Saving the file.
31. dev.off()
```

Output:

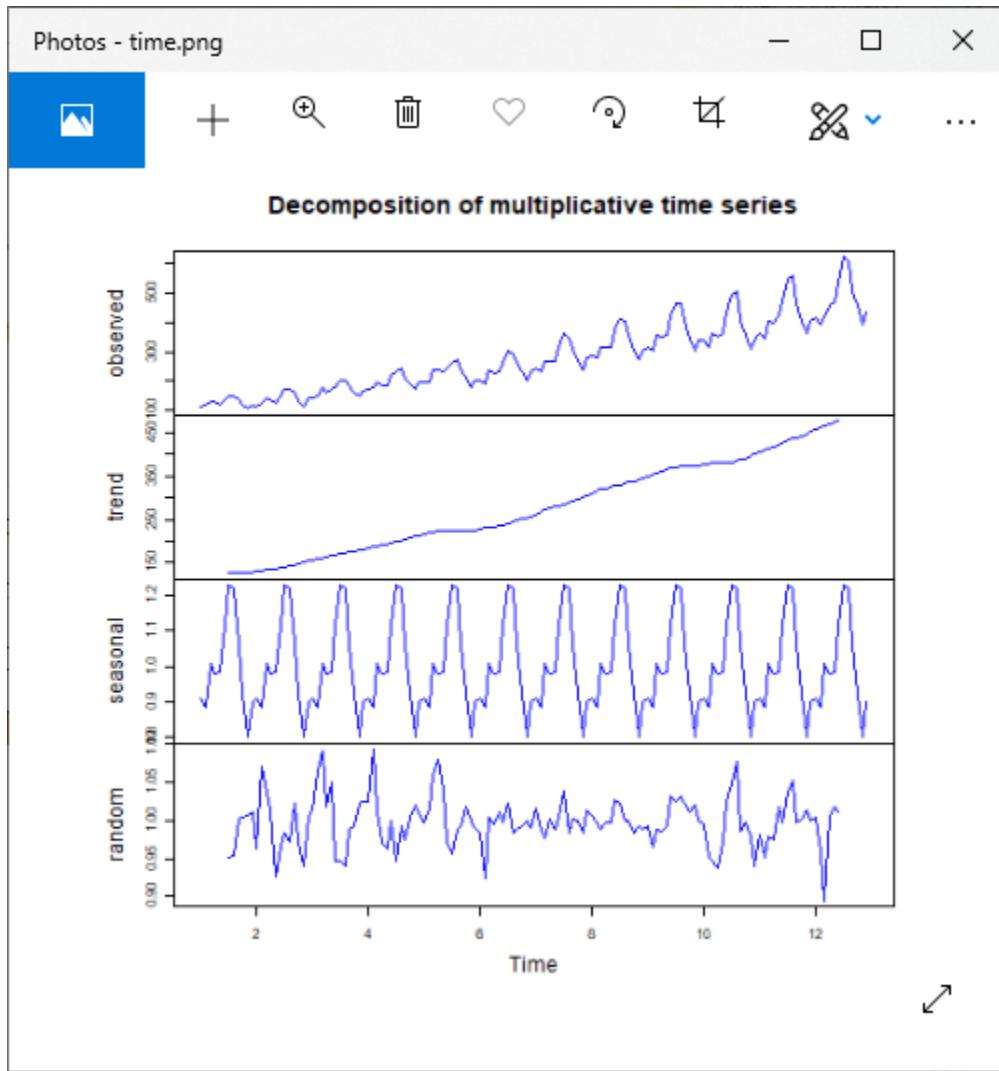


For Multiplicative Model

1. #Importing libraries
2. library(forecast)
3. library(timeSeries)
4. library(fpp)
5. library(Ecdat)
6. #Using Airpassengers data
7. data(AirPassengers)
8. #Creating time series
9. **timeseries.air = AirPassengers**
10. #Detect trend
11. **trend.air = ma(timeseries.air, order = 12, centre = T)**
12. #Detrend of time series
13. **detrend.air=timeseries.air / trend.air**

```
14. #Average the seasonality
15. m.air = t(matrix(data = detrend.air, nrow = 12))
16. seasonal.air = colMeans(m.air, na.rm = T)
17. #Examine the remaining random noise
18. random.air = timeseries.air / (trend.air * seasonal.air)
19. #Reconstruct the original signal
20. recomposed.air = trend.air*seasonal.air*random.air
21. #Decomposed the time series
22. ts.air = ts(timeseries.air, frequency = 12)
23. decompose.decompose.air = decompose(ts.air, "multiplicative")
24.
25. # Giving a name to the file.
26. png(file = "time.png")
27.
28. par(mfrow=c(2,2))
29.
30. plot(as.ts(decompose.air$seasonal),col="blue")
31. plot(as.ts(decompose.air$trend),col="blue")
32. plot(as.ts(decompose.air$random),col="blue")
33. plot(decompose.air,col="blue")
34.
35. # Saving the file.
36. dev.off()
```

Output:



R-Random Forest

The Random Forest is also known as Decision Tree Forest. It is one of the popular **decision tree-based** ensemble models. The accuracy of these models is higher than other decision trees. This algorithm is used for both classification and regression applications.

In a random forest, we create a large number of decision trees, and in each decision tree, every observation is fed. The final output is the most common outcome for each observation. We take a majority vote for each classification model by feeding a new observation into all the trees.

An error estimate is made for cases that were not used when constructing the tree. This is called an **out-of-bag(OOB)** error estimate mentioned as a percentage.

The decision trees are prone to overfitting, and this is the main drawback of it. The reason is that trees, if deepened, are able to fit all types of variations in the data, including noise. It is possible to address this by partial pruning, and the results are often less than satisfactory.

R allows us to create random forests by providing the randomForest package. The randomForest package provides randomForest() function, which helps us to create and analyze random forests. There is the following syntax of random forest in R:

1. randomForest(formula, data)

Example:

Let's start understanding how the randomForest package and its function are used. For this, we take an example in which we used the heart-disease dataset. Let's start our coding section step by step.

1) In the first step, we have to load the three required libraries i.e., ggplot2, cowplot, and randomForest.

1. #Loading ggplot2, cowplot, and randomForest packages
2. library(ggplot2)
3. library(cowplot)
4. library(randomForest)

2) Now, we will use the heart-disease dataset present in <http://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/processed.cleveland.data>. And then, from this dataset, we read the data in CSV format and store it in a variable.

1. #Fetching heart-disease dataset
2. url<- "http://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/processed.cleveland.data"
3. data <- read.csv(url,header=FALSE)

3) Now, we print our data with the help of head() function which prints only the starting six rows as:

1. #Head print six rows of data.
2. head(data)

When we run the above code, it will generate the following output.

Output:

```
> head(data)
  v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14
1 63 1 1 145 233 1 2 150 0 2.3 3 0.0 6.0 0
2 67 1 4 160 286 0 2 108 1 1.5 2 3.0 3.0 2
3 67 1 4 120 229 0 2 129 1 2.6 2 2.0 7.0 1
4 37 1 3 130 250 0 0 187 0 3.5 3 0.0 3.0 0
5 41 0 2 130 204 0 2 172 0 1.4 1 0.0 3.0 0
6 56 1 2 120 236 0 0 178 0 0.8 1 0.0 3.0 0
```

4) From the above output, it is clear that none of the columns are labeled. Now, we name the columns and make these columns labeled in the following way:

```
colnames(data) <- 
  c("age","sex","cp","trestbps","chol","fbs","restecg","thalach","exang","oldpeak","slope",
  "ca","thal","hd")
head(data)
```

Output:

```
> head(data)
  age sex cp trestbps chol fbs restecg thalach exang oldpeak slope ca thal hd
1 63 1 1 145 233 1 2 150 0 2.3 3 0.0 6.0 0
2 67 1 4 160 286 0 2 108 1 1.5 2 3.0 3.0 2
3 67 1 4 120 229 0 2 129 1 2.6 2 2.0 7.0 1
4 37 1 3 130 250 0 0 187 0 3.5 3 0.0 3.0 0
5 41 0 2 130 204 0 2 172 0 1.4 1 0.0 3.0 0
6 56 1 2 120 236 0 0 178 0 0.8 1 0.0 3.0 0
> |
```

5) Let's check the structure of the data with the help of str() function to analyze it better.

1. str(data)

Output:

```

> str(data)
'data.frame': 303 obs. of 14 variables:
 $ age     : num 63 67 67 37 41 56 62 57 63 53 ...
 $ sex      : num 1 1 1 1 0 1 0 0 1 1 ...
 $ cp       : num 1 4 4 3 2 2 4 4 4 4 ...
 $ trestbps: num 145 160 120 130 130 120 140 120 130 140 ...
 $ chol     : num 233 286 229 250 204 236 268 354 254 203 ...
 $ fbs      : num 1 0 0 0 0 0 0 0 1 ...
 $ restecg  : num 2 2 2 0 2 0 2 0 2 2 ...
 $ thalach  : num 150 108 129 187 172 178 160 163 147 155 ...
 $ exang    : num 0 1 1 0 0 0 0 1 0 1 ...
 $ oldpeak  : num 2.3 1.5 2.6 3.5 1.4 0.8 3.6 0.6 1.4 3.1 ...
 $ slope    : num 3 2 2 3 1 1 3 1 2 3 ...
 $ ca       : Factor w/ 5 levels "?","0.0","1.0",...: 2 5 4 2 2 2 4 2 3 2 ...
 $ thal     : Factor w/ 4 levels "?","3.0","6.0",...: 3 2 4 2 2 2 4 4 4 ...
 $ hd       : int 0 2 1 0 0 0 3 0 2 1 ...

```

6) In the above output, we highlight those columns which we will use in our analysis. It is clear from the output that some of the columns are messed up. gender is supposed to be a factor where 0 represents the "Female" and 1 represents the "Male". And cp(chest pain) is also supposed to be a factor where level 1 to 3 represent a different type of pain, and 4 represent no chest pain.

The ca and thal are factors, but one of the levels is "?" when we need it to be NA. We have to clean up the data in our dataset, which are as follows:

1. #Changing the "?" to NAs?
2. data[data=="?"] <- NA
- 3.
4. #Converting the 0's in sex to F and 1's to M
5. data[data\$sex==0]\$sex <- "F"
6. data[data\$sex==1]\$sex <- "M"
- 7.
8. #Converting columns into the factors
9. data\$sex<- as.factor(data\$sex)
10. data\$cp<- as.factor(data\$cp)
11. data\$fbs<- as.factor(data\$fbs)
12. data\$restecg<- as.factor(data\$restecg)
13. data\$exang<- as.factor(data\$exang)
14. data\$slope<- as.factor(data\$slope)
- 15.
16. #ca and thal columns contain? rather than NA. R treats it as a column of string, We correct this assumption by telling R that is a column of integers.
- 17.

```

18. data$ca<- as.integer(data$ca)
19. data$ca<- as.factor(data$ca)
20. data$thal<- as.integer(data$thal)
21. data$thal<- as.factor(data$thal)
22.
23. #Making data hd where 0's represent healthy and 1's to unhealthy.
24. data$hd<- ifelse(test=data$hd==0, yes="healthy", no="Unhealthy")
25. data$hd<- as.factor(data$hd)
26.
27. #Checking structure of data
28. str(data)

```

Output:

```

> str(data)
'data.frame': 303 obs. of 14 variables:
 $ age     : num  63 67 67 37 41 56 62 57 63 53 ...
 $ sex     : Factor w/ 2 levels "F","M": 2 2 2 2 1 2 1 1 2 2 ...
 $ cp      : Factor w/ 4 levels "1","2","3","4": 1 4 4 3 2 2 4 4 4 4 ...
 $ trestbps: num 145 160 120 130 130 120 140 120 130 140 ...
 $ chol    : num 233 286 229 250 204 236 268 354 254 203 ...
 $ fbs     : Factor w/ 2 levels "0","1": 2 1 1 1 1 1 1 1 1 2 ...
 $ restecg : Factor w/ 3 levels "0","1","2": 3 3 3 1 3 1 3 1 3 3 ...
 $ thalach : num 150 108 129 187 172 178 160 163 147 155 ...
 $ exang   : Factor w/ 2 levels "0","1": 1 2 2 1 1 1 1 2 1 2 ...
 $ oldpeak : num 2.3 1.5 2.6 3.5 1.4 0.8 3.6 0.6 1.4 3.1 ...
 $ slope   : Factor w/ 3 levels "1","2","3": 3 2 2 3 1 1 3 1 2 3 ...
 $ ca      : Factor w/ 4 levels "2","3","4","5": 1 4 3 1 1 1 3 1 2 1 ...
 $ thal   : Factor w/ 3 levels "2","3","4": 2 1 3 1 1 1 1 1 3 3 ...
 $ hd      : Factor w/ 2 levels "healthy","Unhealthy": 1 2 2 1 1 1 2 1 2 2 ...
> |

```

7) Now, we are randomly sampling things by setting the seed for the random number generator so that we can reproduce our result.

1. set.seed(42)

8) NWxt, we impute values for the NAs in the dataset with rImpute() function. In the following way:

1. data.imputed<- rImpute(hd~., data=data, iter=6)

Output:

```

> data.imputed <- rfImpute(hd~, data=data, iter=6)
ntree      OOB      1      2
 300: 17.49% 12.80% 23.02%
ntree      OOB      1      2
 300: 16.83% 14.02% 20.14%
ntree      OOB      1      2
 300: 17.82% 13.41% 23.02%
ntree      OOB      1      2
 300: 17.49% 14.02% 21.58%
ntree      OOB      1      2
 300: 17.16% 12.80% 22.30%
ntree      OOB      1      2
 300: 18.15% 14.63% 22.30%
> |

```

9) Now, we build the proper random forest with the help of the `randomForest()` function in the following way:

1. Model<-

```
randomForest(hd~,data=data.imputed,ntree=1000,proximity=TRUE)
```

2. Model

Output:

```

> Model<-randomForest(hd~,data=data.imputed,proximity=TRUE)
> Model

call:
  randomForest(formula = hd ~ ., data = data.imputed, proximity = TRUE)
    Type of random forest: classification
                Number of trees: 500
  No. of variables tried at each split: 3

    OOB estimate of  error rate: 16.83%
Confusion matrix:
             healthy unhealthy class.error
  healthy       142        22   0.1341463
  unhealthy      29       110   0.2086331
> |

```

10) Now, if 500 trees are enough for optimal classification, we will plot the error rates. We create a data frame which will format the error rate information in the following way:

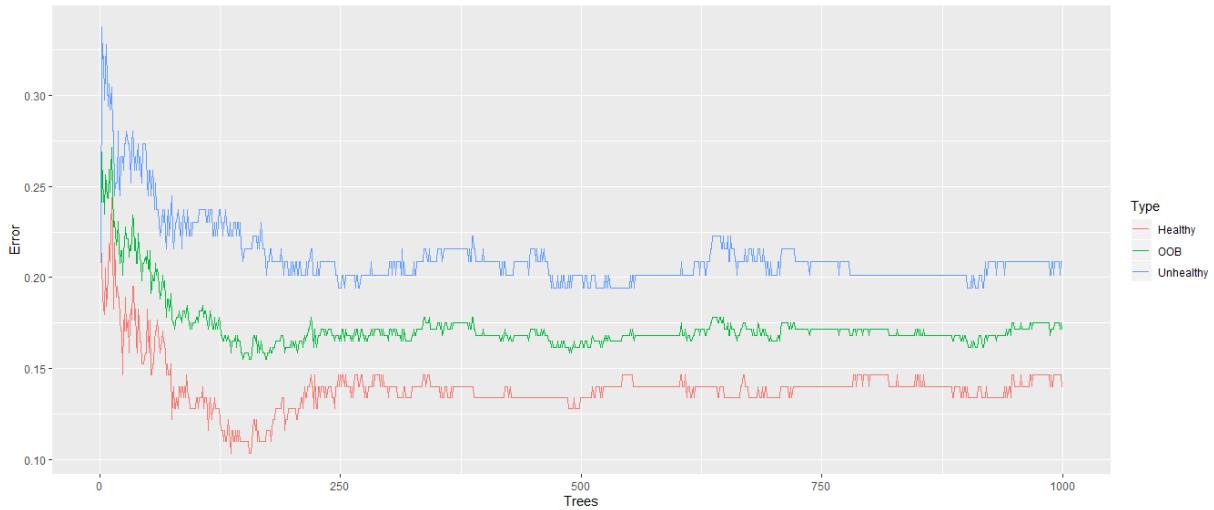
1. `oob_error_data<- data.frame(Trees=rep(1:nrow(Model$err.rate),times=3),Type =rep(c("OOB","Healthy","Unhealthy"),each=nrow(Model$err.rate)),Error=c(Model$err.rate[,"OOB"],Model$err.rate[,"healthy"],Model$err.rate[,"Unhealthy"]))`

11) We call the `ggplot` for plotting error rate in the following way:

- 11) We call the `ggplot` for plotting error rate in the following way:

```
2. ggplot(data=oob_error_data,aes(x=Trees,y=Error))+geom_line(aes(color=Type))
```

Output:



From the above output, it is clear that the error rate decreases when our random forest has more trees.

12) Now, we add 1000 trees and check would the error rate goes down further? So we make a random forest with 1000 trees and find the error rate as we have done before.

```
1. Model<-
```

```
randomForest(hd~.,data=data.imputed,ntree=1000,proximity=TRUE)
```

```
2. Model
```

Output:

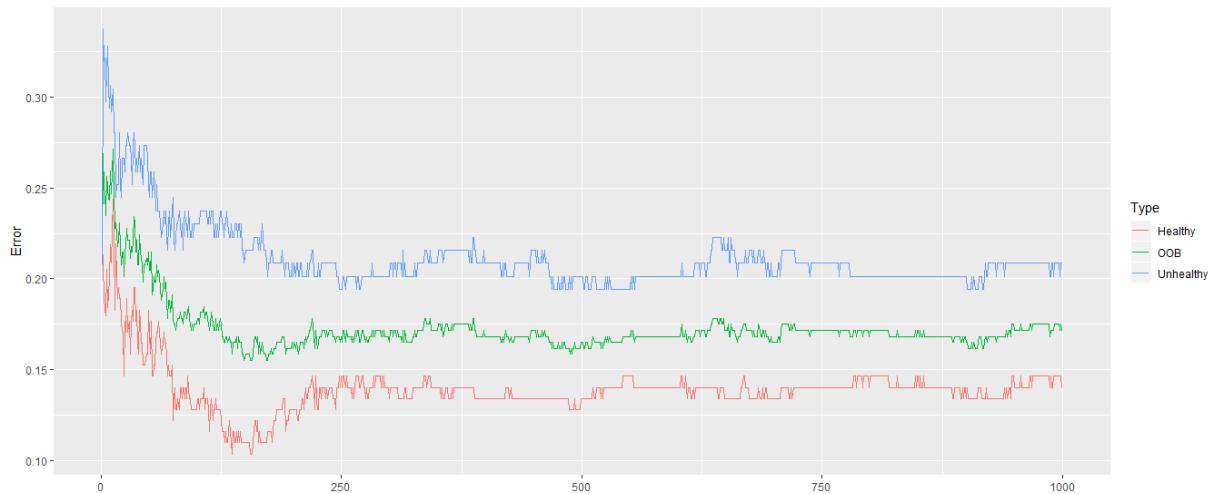
```
> Model<-randomForest(hd~.,data=data.imputed,ntree=1000,proximity=TRUE)
> Model

call:
randomForest(formula = hd ~ ., data = data.imputed, ntree = 1000,      proximity = TRUE)
  Type of random forest: classification
  Number of trees: 1000
No. of variables tried at each split: 3

  oob estimate of  error rate: 17.16%
Confusion matrix:
             healthy unhealthy class.error
healthy       141        23   0.1402439
Unhealthy      29       110   0.2086331
> |
```

```
1. oob_error_data<- data.frame(Trees=rep(1:nrow(Model$err.rate),times=3),Type
=rep(c("OOB","Healthy","Unhealthy"),each=nrow(Model$err.rate)),Error=c(Mo
del$err.rate[,"OOB"],Model$err.rate[,"healthy"],Model$err.rate[,"Unhealthy"]))
2. ggplot(data=oob_error_data,aes(x=Trees,y=Error))+geom_line(aes(color=Type))
```

Output:



From the above output it is clear that the error rate is stabilized.

13) Now, we need to make sure that we are considering the optimal number of variables at each internal node in the tree. This will be done in the following way:

1. #Creating a vector that can hold ten values.
2. oob_values<- vector(length=10)
- 3.
4. #Testing of the different numbers of variables at each step.
5. for(i in 1:10){
6. #Building a random forest for determining the number of variables to try at each step.
7. temp_model<- randomForest(hd~.,data=data.imputed,mtry=i,ntree=1000)
- 8.
9. #Storing OOB error rate.
10. oob_values[i] <- temp_model\$err.rate[nrow(temp_model\$err.rate),1]
11. }
12. oob_values

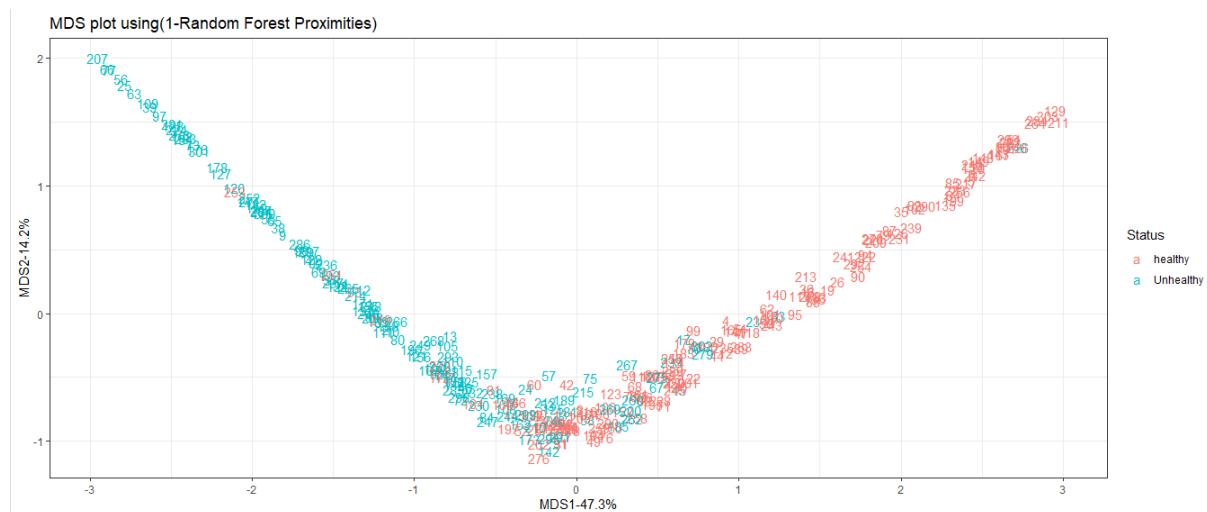
Output:

```
> oob_values <- vector(length=10)
> for(i in 1:10){
+   temp_model <- randomForest(hd~.,data=data.imputed,mtry=i,ntree=1000)
+   oob_values[i] <- temp_model$err.rate[nrow(temp_model$err.rate),1]
+ }
> oob_values
[1] 0.1749175 0.1782178 0.1716172 0.1782178 0.1683168 0.1881188 0.1815182 0.1980198 0.1782178
[10] 0.1848185
```

14) Now, we use the random forest to draw an MDS plot with samples. This will show us how they are related to each other. This will be done in the following way:

1. #Creating a distance matrix with the help of dist() function.
2. distance_matrix<- dist(1-Model\$proximity)
- 3.
4. #Running cmdscale() on the distance matrix.
5. mds_stuff<- cmdscale(distance_matrix,eig=TRUE,x.ret=TRUE)
- 6.
7. #Calculating the percentage of variation in the distance matrix that the X and Y axes account for.
8. mds_var_per<- round(mds_stuff\$eig/sum(mds_stuff\$eig)*100,1)
- 9.
10. #Formatting the data for ggplot() function
11. mds_values<- mds_stuff\$points
12. mds_data<- data.frame(Sample=rownames(mds_values),X=mds_values[,1],Y=mds_values[,2],Status=data.imputed\$hd)
- 13.
14. #Drawing the graph with ggplot() function.
15. ggplot(data=mds_data,aes(x=X,y=Y,label=Sample))+geom_text(aes(color=Status))+theme_bw()+xlab(paste("MDS1-",mds_var_per[1],"%"))+ylab(paste("MDS2-",mds_var_per[2],"%"))+ggtitle("MDS plot using(1-Random Forest Proximities)")

Output:



T-Test in R

In statistics, the T-test is one of the most common test which is used to determine whether the mean of the two groups is equal to each other. The assumption for the test is that both groups are sampled from a normal distribution with equal fluctuation. The null hypothesis is that the two means are the same, and the alternative is that they are not identical. It is known that under the null hypothesis, we can compute a t-statistic that will follow a t-distribution with $n_1 + n_2 - 2$ degrees of freedom.

T-Test in R



In R, there are various types of T-test like **one sample** and **Welch T-test**. R provides a `t.test()` function, which provides a variety of T-tests.

There are the following syntaxes of `t.test()` function for different T-test

Independent 2-group T-test

```
t.test(y~x)
```

here, y is numeric, and x is a binary factor.

Independent 2-group T-test

1. `t.test(y1,y2)`

Here, `y1` and `y2` are numeric.

Paired T-test

1. `t.test(y1,y2,paired=TRUE)`

Here, `y1` & `y2` are numeric.

One sample T-test

1. `t.test(y, mu=3)`

Here, $H_0: \mu = 3$

How to perform T-tests in R

In the T-test, for specifying equal variances and a pooled variance estimate, we set `var.equal=True`. We can also use `alternative="less"` or `alternative="greater"` for specifying one-tailed test.

Let's see how one-sample, paired sample, and independent samples T-test is performed.

One-Sample T-test

One-Sample T-test is a T-test which compares the mean of a vector against a theoretical mean. There is a following formula which is used to compute the T-test :

$$t = \frac{m - \mu}{\frac{s}{\sqrt{n}}}$$

Here,

1. M is the mean.
2. μ is the theoretical mean.
3. s is the standard deviation.
4. n is the number of observations.

For evaluating the statistical significance of the **t-test**, we need to compute the **p-value**. The p-value range starts from 0 to 1, and is interpreted as follow:

- o If the p-value is lower than 0.05, it means we are strongly confident to reject the null hypothesis. So that H_3 is accepted.
- o If the p-value is higher than 0.05, then it indicates that we don't have enough evidence to reject the null hypothesis.

We construct the pvalue by looking at the corresponding absolute value of the t-test.

In R, we use the following syntax of t.test() function for performing a one-sample T-test in R.

1. t.test(x, ?=0)

Here,

1. x is the name of our variable of interest.
2. ? is described by the null hypothesis, which is set equal to the mean.

Example

Let's see an example of One-Sample T-test in which we test whether the volume of a shipment of wood was less than usual($\mu_0=0$).

1. set.seed(0)
2. ship_vol <- c(rnorm(70, mean = 35000, sd = 2000))
3. t.test(ship_vol, mu = 35000)

Output:

```
> set.seed(0)
> ship_vol <- c(rnorm(70, mean = 35000, sd = 2000))
> t.test(ship_vol, mu = 35000)

One Sample t-test

data: ship_vol
t = -0.27702, df = 69, p-value = 0.7826
alternative hypothesis: true mean is not equal to 35000
95 percent confidence interval:
34498.18 35379.45
sample estimates:
mean of x
34938.81
```

Paired-Sample T-test

To perform a paired-sample test, we need two vectors data y1 and y2. Then, we will run the code using the syntax t.test (y1, y2, paired = TRUE).

Example:

Suppose, we work in a large health clinic, and we are testing a new drug Procardia, which aims to reduce high blood pressure. We find 13000 individuals with high systolic blood pressure ($x_{150} = 150$ mmHg, SD = 10 mmHg), and we provide them with Procardia for a month, and then measure their blood pressure again. We find that the

average systolic blood pressure decreased to 144 mmHg with a standard deviation of 9 mmHg.

1. set.seed(2800)
2. pre.treatment <- c(rnorm(2000, mean = 130, sd = 5))
3. post.treatment <- c(rnorm(2000, mean = 144, sd = 4))
4. t.test(pre_Treatment, post_Treatment, paired = TRUE)

Output:

```
> set.seed(2800)
> pre.treatment <- c(rnorm(2000, mean = 130, sd = 5))
> post.treatment <- c(rnorm(2000, mean = 144, sd = 4))
> t.test(pre_Treatment, post_Treatment, paired = TRUE)

  Paired t-test

data: pre_Treatment and post_Treatment
t = 20.789, df = 1999, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 5.593910 6.759259
sample estimates:
mean of the differences
       6.176585
```

Independent-Sample T-test

Depending on the structure of our data and the equality of their variance, the independent-sample T-test can take one of the three forms, which are as follows:

1. Independent-Samples T-test where y1 and y2 are numeric.
2. Independent-Samples T-test where y1 is numeric and y2 is binary.
3. Independent-Samples T-test with equal variances not assumed.

There is the following general form of t.test() function for the independent-sample t-test:

1. t.test(y1,y2, paired=FALSE)

By default, R assumes that the versions of y1 and y2 are unequal, thus defaulting to Welch's test. For toggling this, we set the flag var.equal=TRUE.

Let's see some examples in which we test the hypothesis. In this hypothesis, Clevelanders and New Yorkers spend different amounts for eating outside on a monthly basis.

Example 1: Independent-Sample T-test where y1 and y2 are numeric

1. set.seed(0)
2. Spenders.Cleve <- rnorm(50, mean = 300, sd = 70)
3. Spenders.NY <- rnorm(50, mean = 350, sd = 70)
4. t.test(Spenders.Cleve, Spenders.NY, var.equal = TRUE)

Output:

```
> set.seed(0)
> Spenders.Cleve <- rnorm(50, mean = 300, sd = 70)
> Spenders.NY <- rnorm(50, mean = 350, sd = 70)
> t.test(Spenders.Cleve, Spenders.NY, var.equal = TRUE)

Two Sample t-test

data: Spenders.Cleve and Spenders.NY
t = -4.0115, df = 98, p-value = 0.0001179
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-74.47029 -25.17621
sample estimates:
mean of x mean of y
301.6752 351.4984
```

Example 2: Where y1 is numeric and y2 are binary

1. set.seed(0)
2. Spenders.Cleve <- rnorm(50, mean = 300, sd = 70)
3. Spenders.NY <- rnorm(50, mean = 350, sd = 70)
4. Amount.Spent <- c(Spenders.Cleve, Spenders.NY)
5. city.name <- c(rep("Cleveland", 50), rep("New York", 50))
6. t.test(Amount.Spent ~ city.name, var.equal = TRUE)

Output:

```
> set.seed(0)
> Spenders.Cleve <- rnorm(50, mean = 300, sd = 70)
> Spenders.NY <- rnorm(50, mean = 350, sd = 70)
> Amount.Spent <- c(Spenders.Cleve, Spenders.NY)
> city.name <- c(rep("Cleveland", 50), rep("New York", 50))
> t.test(Amount.Spent ~ city.name, var.equal = TRUE)

Two Sample t-test

data: Amount.Spent by city.name
t = -4.0115, df = 98, p-value = 0.0001179
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-74.47029 -25.17621
sample estimates:
mean in group Cleveland mean in group New York
301.6752                 351.4984
```

Example 3: With equal variance not assumed

1. **set.seed(0)**
2. **Spenders.Cleve <- rnorm(50, mean = 300, sd = 70)**
3. **Spenders.NY <- rnorm(50, mean = 350, sd = 70)**
4. **t.test(Spenders.Cleve, Spenders.NY, var.equal = FALSE)**

Output:

```
> set.seed(0)
> Spenders.Cleve <- rnorm(50, mean = 300, sd = 70)
> Spenders.NY <- rnorm(50, mean = 350, sd = 70)
> t.test(Spenders.Cleve, Spenders.NY, var.equal = FALSE)

Welch Two Sample t-test

data: Spenders.Cleve and Spenders.NY
t = -4.0115, df = 97.565, p-value = 0.0001183
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-74.47166 -25.17483
sample estimates:
mean of x mean of y
301.6752   351.4984
```

Chi-Square Test

The **Chi-Square Test** is used to analyze the frequency table (i.e., contingency table), which is formed by two categorical variables. The chi-square test evaluates whether there is a significant relationship between the categories of the two variables.

The Chi-Square Test is a statistical method which is used to determine whether two categorical variables have a significant correlation between them. These variables should be from the same population and should be categorical like- Yes/No, Red/Green, Male/Female, etc.

R provides **chisq.test()** function to perform chi-square test. This function takes data as an input, which is in the table form, containing the count value of the variables in the observation.

Chi-Square Test in R



In R, there is the following syntax of chisq.test() function:

1. `chisq.test(data)`

Let's see an example in which we will take the Cars93 data present in the "Mass" library. This data represents the sales of different models of cars in the year 1993.

Data:

1. `library("MASS")`
2. `print(str(Cars93))`

Output:

```

> library("MASS")
> print(str(cars93))
'data.frame': 93 obs. of 27 variables:
 $ Manufacturer : Factor w/ 32 levels "Acura","Audi",...: 1 1 2 2 3 4 4 4 4 5 ...
 $ Model        : Factor w/ 93 levels "100","190E","240",...: 49 56 9 1 6 24 54 74 73 35 ...
 $ Type         : Factor w/ 6 levels "Compact","Large",...: 4 3 1 3 3 3 2 2 3 2 ...
 $ Min.Price    : num 12.9 29.2 25.9 30.8 23.7 14.2 19.9 22.6 26.3 33 ...
 $ Price        : num 15.9 33.9 29.1 37.7 30 15.7 20.8 23.7 26.3 34.7 ...
 $ Max.Price   : num 18.8 38.7 32.3 44.6 36.2 17.3 21.7 24.9 26.3 36.3 ...
 $ MPG.city     : int 25 18 20 19 22 22 19 16 19 16 ...
 $ MPG.highway  : int 31 25 26 26 30 31 28 25 27 25 ...
 $ AirBags      : Factor w/ 3 levels "Driver & Passenger",...: 3 1 2 1 2 2 2 2 2 2 ...
 $ DriveTrain   : Factor w/ 3 levels "4WD","Front",...: 2 2 2 3 2 2 3 2 2 ...
 $ Cylinders    : Factor w/ 6 levels "3","4","5","6",...: 2 4 4 4 2 2 4 4 4 5 ...
 $ EngineSize   : num 1.8 3.2 2.8 2.8 3.5 2.2 3.8 5.7 3.8 4.9 ...
 $ Horsepower   : int 140 200 172 172 208 110 170 180 170 200 ...
 $ RPM          : int 6300 5500 5500 5500 5700 5200 4800 4000 4800 4100 ...
 $ Rev.per.mile : int 2890 2335 2280 2535 2545 2565 1570 1320 1690 1510 ...
 $ Man.trans.avail: Factor w/ 2 levels "No","Yes": 2 2 2 2 1 1 1 1 1 ...
 $ Fuel.tank.capacity: num 13.2 18 16.9 21.1 21.1 16.4 18 23 18.8 18 ...
 $ Passengers   : int 5 5 5 6 4 6 6 6 5 6 ...
 $ Length        : int 177 195 180 193 186 189 200 216 198 206 ...
 $ wheelbase     : int 102 115 102 106 109 105 111 116 108 114 ...
 $ width         : int 68 71 67 70 69 69 74 78 73 73 ...
 $ Turn.circle   : int 37 38 37 37 39 41 42 45 41 43 ...
 $ Rear.seat.room: num 26.5 30 28 31 27 28 30.5 30.5 26.5 35 ...
 $ Luggage.room  : int 11 15 14 17 13 16 17 21 14 18 ...
 $ weight        : int 2705 3560 3375 3405 3640 2880 3470 4105 3495 3620 ...
 $ Origin        : Factor w/ 2 levels "USA","non-USA": 2 2 2 2 2 1 1 1 1 1 ...
 $ Make          : Factor w/ 93 levels "Acura Integra",...: 1 2 4 3 5 6 7 9 8 10 ...
NULL
> |

```

Example:

1. # Loading the Mass library.
2. library("MASS")
3. # Creating a data frame from the main data set.
4. car_data<- data.frame(Cars93\$AirBags, Cars93>Type)
5. # Creating a table with the needed variables.
6. **car_data = table(Cars93\$AirBags, Cars93>Type)**
7. print(car_data)
8. # Performing the Chi-Square test.
9. print(chisq.test(car_data))

Output:

```
> # Loading the Mass library.  
> library("MASS")  
>  
> # Creating a data frame from the main data set.  
> car_data <- data.frame(Cars93$AirBags, Cars93>Type)  
>  
> # Create a table with the needed variables.  
> car_data = table(Cars93$AirBags, Cars93>Type)  
> print(car_data)  
  
          Compact Large Midsize Small sporty Van  
Driver & Passenger    2     4      7     0     3     0  
Driver only            9     7     11     5     8     3  
None                  5     0      4    16     3     6  
>  
> # Perform the Chi-Square test.  
> print(chisq.test(car_data))  
  
Pearson's chi-squared test  
  
data: car_data  
X-squared = 33.001, df = 10, p-value = 0.0002723  
  
Warning message:  
In chisq.test(car_data) : chi-squared approximation may be incorrect  
> |
```